Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Tesis de Licenciatura

Análisis de Diseño Usando Traits sin Subclasificación

Autor:
Diego Campodónico
LU 561/03

Director:
Lic. Hernán Wilkinson

Buenos Aires, diciembre de 2011

Resumen

La herencia simple, usada en los lenguajes orientados a objetos, es un modelo de organización del comportamiento de los entes[1] de la realidad. A pesar de ser un mecanismo de construcción útil, la herencia simple tiene problemas de representación. Ante la necesidad de compartir comportamiento, las herramientas de modelado que brinda no ofrecen suficientes mecanismos de representación, y esto fuerza a la repetición de código o a la necesidad de compartir código innecesariamente.
Los traits surgen como un mecanismo complementario a la herencia simple, para solucionar los problemas que esta tiene.
En trabajos relacionados, se han utilizado traits junto con subclasificación para solucionar los problemas antes mencionados.
Sin embargo, la utilización de traits y subclasificación plantea una dificultad adicional al momento de diseñar: decidir si compartir comportamiento utilizando traits o subclasificación. Además, la utilización de dos mecanismos de abstracción diferentes agrega complejidad al modelo.
El objetivo de este trabajo es determinar las ventajas y desventajas de utilizar un nuevo modelo en el cual se usa únicamente el concepto de trait para compartir comportamiento entre objetos, excluyendo la subclasificación.
Para cumplir el objetivo, se creará un nuevo modelo de la jerarquía de Collections de Smalltalk utilizando solo traits y sus características se contrastarán contra el modelo actual basado en subclasificación. La comparación se centrará en cuestiones de representación, modelado y desarrollo.
Basándonos en los resultados obtenidos, el nuevo modelo soluciona los problemas de la herencia simple anteriormente mencionados, pero por otro lado genera un modelo con mayor complejidad.

[1] Para ejemplos, ver: http://objectmodels.blogspot.com/2007/06/relacin-entre-objetos-y-entes-del.html

Agradecimientos

Quiero agradecer a toda mi familia por el apoyo constante, especialmente a Ana, mi mamá, Silvana y Flor, mis hermanas, y Diego, mi papá.
Gracias a Agostina, mi novia, por darme ánimo y por su ayuda.
Agradezco a Hernán Wilkinson por todas sus enseñanzas, por su guía para la elección de este trabajo, y por su dirección.
También a Gabriela Arévalo por el gran esfuerzo en sus correcciones.
A Nicolás Font, por su interés en mi trabajo y por darme sus sugerencias y opiniones.
A mis amigos, y compañeros de trabajo, por su comprensión e interés en mi trabajo.

Índice

1 Introducción

En los lenguajes orientados a objetos, la herencia simple es el mecanismo más utilizado para organizar el comportamiento de los objetos. Sin embargo, existen escenarios que la herencia simple representa incorrectamente.
Diferentes herramientas de modelado, como herencia múltiple, mixins y traits han surgido para solucionar estos problemas.
En esta tesis utilizaremos traits, que son un mecanismo complementario a la herencia simple, y surgen para solucionar los problemas que esta tiene. Los mismos permiten compartir comportamiento entre objetos, y es posible componerlos entre sí.
En trabajos relacionados[3][4][5][15], se ha estudiado el uso de traits como complemento de la subclasificación. Pero la utilización de dos mecanismos diferentes para la organización del comportamiento presenta una dificultad adicional al diseñador a la hora de tomar decisiones de modelado.

1.1 Objetivos del Trabajo de Investigación

El principal objetivo de la tesis es realizar un análisis profundo generando un modelo donde la manera de compartir comportamiento entre objetos sea a través de composición de traits en vez de utilizar subclasificación.
Al modelar sin subclasificación, y utilizando traits, estudiaremos qué representa un trait en la realidad y así encontrar axiomas para la utilización de los mismos.
Para esto, utilizando un ambiente orientado a objetos se creará un nuevo modelo de la jerarquía de Collections utilizando la idea anteriormente explicada, y luego se comparará este nuevo modelo con el actual.
La jerarquía de Collection provee el comportamiento para manipular y operar sobre una colección de objetos, a los que se los denominan elementos.
Existen una gran variedad de tipos de colecciones dependiendo de sus características, como por ejemplo tamaño fijo, variable, ordenadas, no ordenadas.
Por el frecuente uso de las colecciones dentro del entorno, la jerarquía que las reifica es una de las más importantes, y su buena implementación puede beneficiar mucho al usuario del entorno.

1.2 Motivación

Las razones para la aplicación de los traits como el único mecanismo de construcción son:

El paradigma de clasificación tiene algunos problemas, entre los cuales se encuentra el hecho de que el envío de mensajes con *super* agrega complejidad. En muchos casos es difícil entender qué representa y cuándo utilizar ese mecanismo. Y además, muchas veces se desconoce o se comprende mal su funcionamiento.
Al tener un modelo que sólo utiliza traits para compartir comportamiento, no es necesario el uso de *super* y por lo tanto no tenemos los problemas antes mencionados.

En modelos híbridos, muchas veces es difícil decidir cuándo utilizar traits o subclasificación para compartir comportamiento. De nuevo, este problema no estaría presente si utilizamos el modelo propuesto en esta tesis.

1.3 Implementación

Para implementar este nuevo modelo basado en traits sin subclasificación se utilizará Smalltalk como lenguaje de programación, porque:

Implementa traits
Cuenta con una sintaxis simple y declarativa
Es reflexivo, esto significa que permite acceder y trabajar con el metamodelo en el cual está escrito.
Es metacircular, lo que significa que está escrito en el mismo lenguaje que interpreta.
Permite comprender el metamodelo de traits, así como también el metamodelo de subclasificación.

En particular se utilizará Pharo [13], una implementación de Smalltalk que tiene una implementación de traits y utiliza licencias MIT [14]
La versión de Pharo utilizada es la 1.1.

Elegimos la jerarquía de Collections para crear un nuevo modelo utilizando únicamente traits. Los motivos de esta elección son:

- La importancia de las colecciones, ya que las mismas son utilizadas para modelar casi cualquier dominio de la realidad.
- Las colecciones poseen una gran variedad de características y además estas características se combinan de diferentes maneras en las mismas, lo cual genera un dominio complejo de modelar.
- Poseen un protocolo bien definido y estandarizado
- Se han realizaron otros trabajos relacionados sobre Collections

En este trabajo nos basamos en el estándar ANSI de Collections por lo siguiente:

- Define los mensajes principales del protocolo de Collections, acotando el dominio y desambiguándolo

- Permite aplicar las conclusiones del trabajo a otras implementaciones de Smalltalk, y realizar comparaciones con otros trabajos basados en el estándar.

2 Marco Teórico

La aplicación de traits implica el desarrollo de un nuevo modelo que solucione los problemas que presenta la herencia simple. En este capítulo detallaremos cuáles son los principales mecanismos que proveen los sistemas orientados a objetos e identificaremos en cada caso los sus problemas.

2.1 Paradigma de objetos

Los diferentes paradigmas de programación como el funcional, el lógico, el estructurado y el paradigma de objetos buscan poder representar los dominios de un problema de la realidad con modelos computacionales. Cada uno de éstos brinda un conjunto de mecanismos que permiten comprender y representar el dominio de la problemática dada. En el caso del paradigma funcional las herramientas son funciones, en el lógico son fórmulas, en el estructurado son estructura de datos y operaciones, mientras que en el de objetos son objetos y mensajes.

Un modelo computacional[2] de un dominio de la realidad busca acortar la brecha semántica entre el modelo obtenido y el dominio. Esto otorga varias ventajas, una de las más importantes es la flexibilidad ante los cambios naturales que tiene un sistema cuando evoluciona. Dado que la realidad cambia constantemente, cuanto mayor es la proximidad del modelo al dominio del problema, más fácil será adaptarlo a la nueva realidad.

En particular, el paradigma de programación orientada a objetos tiene dos elementos para comprender la realidad y expresarla en un modelo computacional: objetos y mensajes.

Dentro del paradigma orientado a objetos un programa se define como: “objetos que colaboran enviándose mensajes”.

2.2 Clasificación

El aprendizaje del dominio del problema es fundamental para comprender cuales son las entidades del mismo y cómo interactúan entre sí.

[2] Por computacional se entiende que puede ser simulado en una maquina de Turing[18] y terminar.

La forma de aprendizaje del ser humano está fuertemente relacionada con la habilidad de la mente de unificar experiencias similares. Por ejemplo, cuando un ser humano observa un auto, esta experiencia es tomada tanto en forma literal como abstracta. Por un lado está el hecho de ver el auto y reconocerlo como tal y, por el otro, ver el conjunto de características que tiene la idea de auto.

Esta forma en la que el ser humano va aprendiendo y comprendiendo puede verse como una clasificación de los diferentes entes de la realidad con los que va experimentando.

Esta idea fue tomada en SIMULA 67 [8] y luego en Smalltalk [9] para representar conocimiento a través del concepto de “Clase”, que es la reificación[3] de lo que antes llamamos ideas. Además las clases tienen la responsabilidad de generar instancias o manifestaciones de las ideas que representan.

2.3 Modelos de Subclasificación

Las clases modelan el comportamiento de los objetos, y mediante la subclasificación, podemos factorizar comportamiento común entre objetos.

Con la herencia, una clase obtiene tanto el comportamiento como la estructura de su superclase. Es importante destacar que la subclasificación no es esencial al paradigma de objetos. Esto lo demuestra la existencia de lenguajes como Self [10] que implementan el paradigma sin contar con clases.

A continuación se realizará una breve descripción de los diferentes modelos de subclasificación existentes hasta el momento. También se detallan los problemas que tienen los mismos y se finaliza la sección con la presentación de Traits y cómo éste soluciona los inconvenientes que hasta el momento no habían sido resueltos.

2.3.1 Herencia Simple

Descripción: Este modelo de herencia es el más simple, permite a una clase subclasificar como máximo de una superclase.

Problemas: Más allá de ser un modelo bien aceptado por la comunidad, en jerarquías complejas se hace más difícil, y a veces imposible, la tarea de factorizar el comportamiento común compartido entre las clases, produciendo así código duplicado.

[3] Reificación en el ámbito de la programación significa representar un concepto en el paradigma en el que estamos trabajando, y utilizando los elementos del mismo.

Como ejemplo se puede citar las clases de Stream en Smalltalk: ReadStream, WriteStream y ReadWriteStream. Como sugiere su nombre ReadWriteStream contiene las características provistas por ReadStream y WriteStream, sin embargo el modelo de herencia simple, permite que ReadWriteStream herede solamente de una de estas clases. En Smalltalk, ReadWriteStream hereda de WriteStream, y duplica los métodos de ReadStream.

2.3.2 Herencia Múltiple

Descripción: La herencia múltiple permite heredar a una clase de una o más superclases. Este mecanismo provee mejor reúso de código y mayor flexibilidad con respecto a la herencia simple. De todos modos, la herencia múltiple utiliza la noción de clase de manera contradictoria: como creador de instancias y como la unidad mínima de reúso de código, lo que genera una ambigüedad que surge de heredar por diferentes caminos.
Esto trae dos problemas, en primer lugar el conflicto de mensajes y en segundo lugar el conflicto en los colaboradores internos. Cuando hablamos de conflicto, nos referimos a una repetición en algunas de las superclases. En general el primero es resuelto de manera explícita especificando qué camino se debe tomar, pero el segundo es más complejo ya que, no siempre tenemos en claro si debería haber una única variable de instancia por cada superclase o si bastaría con tener una única para ambas.
Un caso particular de este problema es el “Diamond Problem” [11], el mismo se produce cuando una clase hereda de una única clase base por diferentes caminos.

2.4 Herencia por Mixins

Descripción: Un Mixin [12] es una especificación de una subclase que puede ser aplicada a varias clases padres para poder extenderlas con el mismo comportamiento.

Proveen mayor capacidad de reúso de código que la herencia simple, manteniendo la simplicidad de la misma. De todas formas, no funciona correctamente cuando se desea componer más de un mixin para una misma clase. A continuación se plantean en detalle los problemas de este modelo de herencia.

Problemas

Orden Total
La composición entre mixins es lineal, todos los mixins usados por una clase deben ser heredados de a uno por vez. Dado esto, el último mixin que se componga sobrescribirá todas las características de los mixins previamente compuestos en la misma jerarquía. El problema con esto radica en que puede no existir un ordenamiento adecuado para obtener el comportamiento deseado.

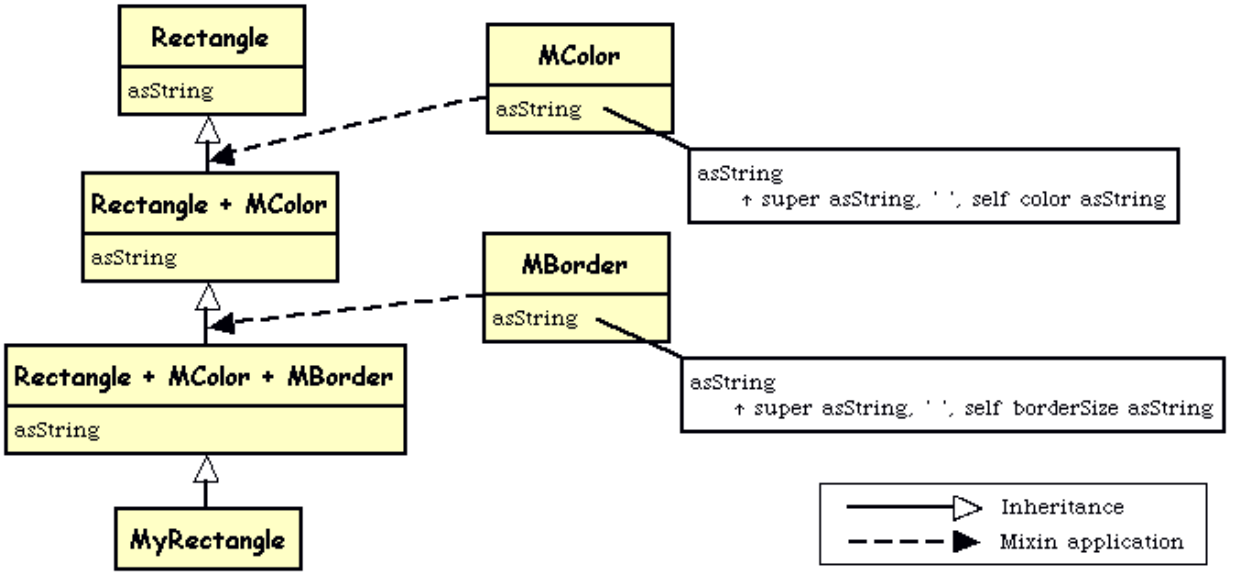

Figura 1 [1] - **Ejemplo del Problema de Mixins**

Observamos que en la Figura 1 el código que interconecta los mixins está especificado en el mixin MBorder. La clase compuesta MyRectangle no puede acceder a la implementación de asString en el mixin MColor y ni a la de la clase Rectangle. Las clases que contienen + en sus nombres son clases intermedias generadas por la aplicación de mixins.

Otro problema que presentan los mixins es el glue code disperso.
La entidad sobre la que se componen los mixins no posee control total en la forma en que los mixins son aplicados. Esto puede traer diversos problemas y para resolverlos es necesario escribir código dentro de los mixins, o agregar nuevos mixins, y hasta se puede requerir utilizar dos veces el mismo mixin en la misma jerarquía.

Los Mixins también generan jerarquías frágiles.
Debido a la linealidad y a los escasos recursos para resolver conflictos, al componer múltiples mixins se obtienen como resultado jerarquías frágiles (poco flexibles) con respecto al cambio.

2.5 Traits

Un trait[1][2] es una unidad de reúso de código para clases que, a su vez, puede componerse a partir de otros traits.
Hoy en día el uso de este concepto se está imponiendo como una nueva forma de pensar los modelos computacionales dentro del paradigma de objetos. Los trabajos realizados hasta el momento[3][4][5] han demostrado que el correcto uso de esta técnica provee estructura, modularidad y reusabilidad junto a las clases. Y de esta manera, ayuda también a encontrar el balance entre la reusabilidad y la comprensión del modelo.

Las siguientes propiedades de los traits solucionan los problemas y limitaciones antes mencionadas:

- Proveen un conjunto de métodos que implementan comportamiento.
- Pueden requerir también, un conjunto de métodos que son utilizados como parámetros del comportamiento que provee.
- No tienen estado, por lo tanto los métodos que provee no acceden a variables de estado directamente.
- Las clases y los traits pueden ser compuestos a partir de otros traits, pero el orden en que se realice dicha composición es irrelevante,
- Los métodos en conflicto deben ser resueltos explícitamente.
- La semántica de la clase no se ve alterada por la composición de los traits. Es decir, que componer una clase con un trait es equivalente a definir todo el comportamiento provisto por el trait dentro de la clase.
- De la misma manera, la composición de un trait con otro no afecta la semántica del primero.

Una gran diferencia entre traits y los modelos como Herencia Múltiple o Mixins es que no se basan en el operador herencia para realizar la composición, sino que tiene sus propios operadores complementarios a los de la herencia simple. Un trait es solamente un conjunto de métodos y no tiene relación con ninguna jerarquía de clases. Además, a diferencia de los mixins, a los traits no es necesario asignarle un orden a la composición.

3 Metodología de Trabajo

El desarrollo del presente trabajo requiere de la definición de una metodología de trabajo para asegurar la calidad y correctitud del nuevo modelo de Collections, así como también optimizar los tiempos de desarrollo. En este capítulo se explicarán los motivos de la elección de la citada metodología y se introducirá el concepto de Test Driven Development, ya que es un método utilizado en la implementación del nuevo modelo.

3.1 Test Driven Development

Test Driven Development (TDD)[17] o Desarrollo Guiado por Pruebas es una metodología la cual busca hacer explícito un requerimiento a través de uno o varios test, y que a partir de estos tests se genere el código que los satisfaga. De esta forma se asegura la correctitud de la implementación del requerimiento.
Esta metodología, también involucra ciclos de refactorización de código. Es decir, una vez que el código satisface el test, es necesaria una revisión del mismo de forma tal que además de pasar los tests, el código sea un buen modelo de la realidad. Estos ciclos buscan asegurar la calidad del código.
De esta manera, se promueve una rápida interacción entre el sistema y el desarrollador, evitando que se pase demasiado tiempo programando sin ejecutar el código.
Los pasos son los siguientes:

1. **Escribir un test**:
   El mismo se basa en algún requerimiento.

2. **Correr el test escrito**:
   Si el test no falla, significa que el requerimiento ya es contemplado por el modelo o bien, el test es erróneo. En el caso de que no sea un error del test escrito, y que el modelo efectivamente pasa el test, entonces se debe volver al paso anterior. Si el test falla se continúa con el paso siguiente.

3. **Hacer funcionar el test**:
   Se debe escribir código de manera tal que el modelo pase el test.

4. **Correr todos los tests**:
   Este paso nos asegura que los cambios efectuados no rompen tests anteriores. En caso de que algún test falle es necesario volver al paso 2.

5. **Mejorar el Diseño**:
   Una vez que se pasan todos los tests es necesario revisar el modelo para que sea un buen modelo. Por cada mejora que se haga es necesario verificar que los tests siguen corriendo. Una vez finalizado el ciclo: Refactoring - Tests, se puede volver a empezar desde el paso 1.

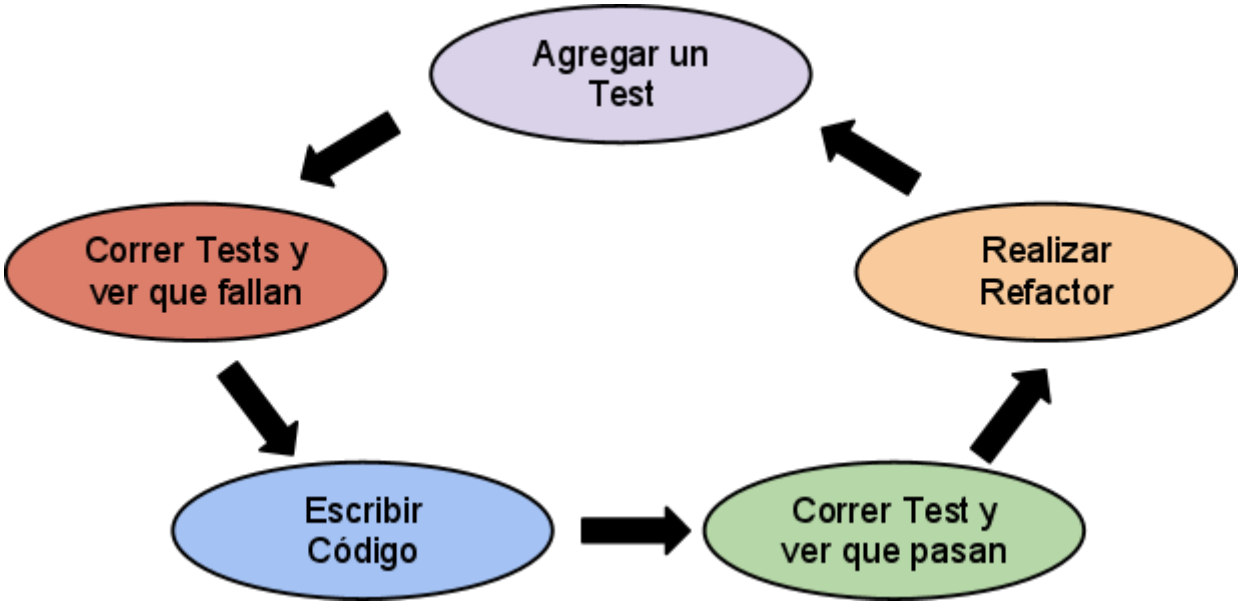

Figura 2 - Ciclo de Desarrollo Utilizando TDD

La Figura 2 muestra gráficamente los pasos anteriormente descriptos.

3.2 Plan de Desarrollo

El modelo basado en traits necesita de un plan de desarrollo considerando los siguientes factores:

- Uno es la necesidad de coexistan ambos modelos de forma tal que puedan ser comparados. Esto hace que no sea una refactorización de código común, ya que no se está modificando el modelo, sino que se está creando otro.
- Otro factor a tener en cuenta es el hecho de que solo vamos a refactorizar una parte del protocolo de Collection, y ese es el protocolo definido en el ANSI.

Teniendo en cuenta estos factores, los pasos del plan son los siguientes:

1. **Escribir tests del protocolo ANSI para el actual modelo**
   El hecho de que el modelo deba respetar el protocolo del ANSI, implica que se deben escribir nuevos tests, ya que los actuales lo respetan parcialmente. La sección Tests del Protocolo ANSI Utilizando Traits Sin Subclasificación explica el desarrollo de estos

tests.

2. **Desarrollo del Nuevo Modelo Utilizando TDD**
   Una vez escritos los tests para el modelo actual de Collection, podemos comenzar con el desarrollo del nuevo modelo utilizando TDD.
   La idea de los tests escritos en el paso anterior es que sirvan también para hacer TDD en este paso, o sea reutilizarlos realizando cambios mínimos.
   De este modo estaríamos haciendo TDD con la diferencia de que ya contamos con todos los tests escritos, pero podemos ir generando código de manera tal de hacer funcionar uno a la vez. Por lo tanto el resultado es el mismo.
   Además, es necesario definir un proceso detallado y ordenado en cuanto a la implementación.
   Dado que estamos copiando los algoritmos de los métodos de Collections de Pharo, se debe tener en cuenta por qué clases y métodos es necesario empezar, ya que las clases dependen unas de otras como así también los métodos. La Figura 3 muestra los pasos a seguir en el desarrollo, que son descriptos a continuación:
   a. Seleccionar una clase concreta de la jerarquía de Collection que no dependa de otra clase concreta todavía no implementada y crearla.
   b. Crear la clase de test correspondiente a la clase anterior, reutilizando los traits de test del protocolo ya creados.
   c. Seleccionar un método de la clase anterior que no dependa de métodos todavía no implementados e implementarlo. Siempre debemos comenzar por métodos de creación de instancias.
   d. Seleccionar el o los tests correspondientes al método anterior e implementar los requerimientos de los que dependa, y luego correrlo.
   e. En caso de que se pase el test, evaluar un posible refactoring del método. Debemos analizar si se puede compartir el comportamiento con otra clase a través de un trait o no.
   f. Continuar con el proceso seleccionando métodos y clases hasta terminar con la jerarquía.

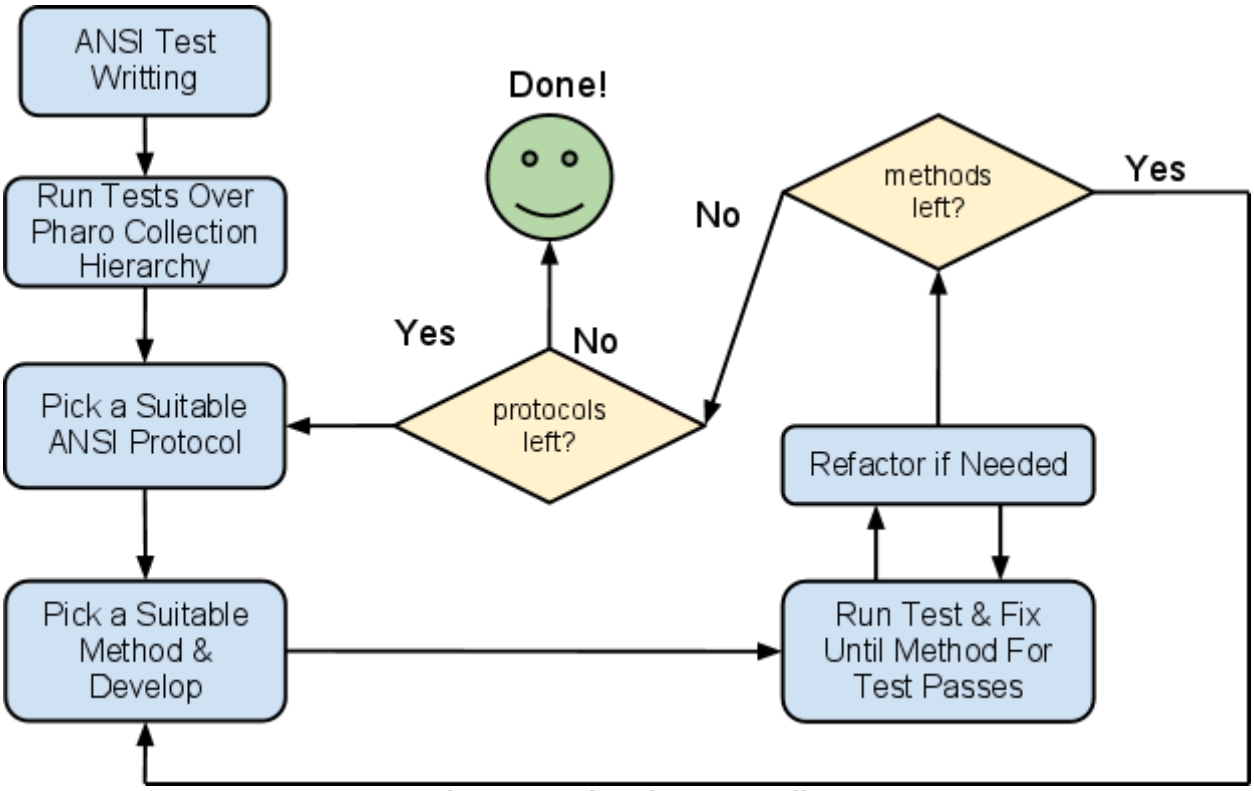

Figura 3 - Plan de Desarrollo

4 Desarrollo del Nuevo Modelo de Collections Utilizando Traits sin Subclasificación

Siguiendo la metodología de trabajo descripta anteriormente, crearemos primero un modelo de tests, y luego el nuevo modelo de Collections.
En esta sección se describirá en detalle todo el desarrollo del nuevo modelo, incluyendo, tanto la implementación de los tests, como el desarrollo del nuevo modelo en sí mismo.

4.1 Tests del Protocolo ANSI Utilizando Traits Sin Subclasificación

Dado que los tests de Collection que están implementados respetan un protocolo propio de la implementación particular de Smalltalk, es necesario escribir nuevos tests que respeten el protocolo ANSI.
El protocolo de la implementación de Pharo respeta en su gran mayoría al protocolo ANSI de Collection, por lo tanto los tests existentes nos servirán como punto de partida para escribir los nuevos. Existen algunos métodos del ANSI que Pharo no implementa, los mismos se especifican en la sección: Métodos del protocolo Collection del ANSI no Implementados en Pharo
Se decidió escribir los tests bajo la misma consigna con la cual se va a crear el nuevo modelo de Collection, es decir utilizando traits sin utilizar subclasificación.
La implementación de los tests de Collections en Pharo utiliza mucho de los traits, esto hace que podamos reutilizar los mismos en nuestra nueva suite de tests.

4.1.1 Modelo

Para explicar el modelo utilizado en los tests de Collections es necesario conocer los protocolos de Collection definidos en el ANSI así como también la relación entre ellos. Una breve explicación se encuentra en la sección: Relaciones entre Protocolos de Collection del ANSI.
Existen dos entidades principales en la definición del protocolo Collection del ANSI: protocolos y relaciones “cumple con”.

Además los protocolos pueden ser “concretos” o “abstractos”[4].
Teniendo en cuenta que cada protocolo concreto es representado en el sistema por una clase concreta, debemos tener un test para esa clase. Cada uno de los tests de clases concretas los representaremos con clases.
Por otro lado, cada protocolo concreto, debe cumplir con protocolos abstractos. Vamos a utilizar entonces a los traits para representar protocolos abstractos, y la relación “usa” que existe entre clases y traits para representar la relación “cumple con”.
La relación “cumple con” es una relación uno a muchos, y se representa con la relación “usa” ya que la composición de traits permite hacer uno de más de uno.
La Figura 4 muestra la forma de representar estas características:

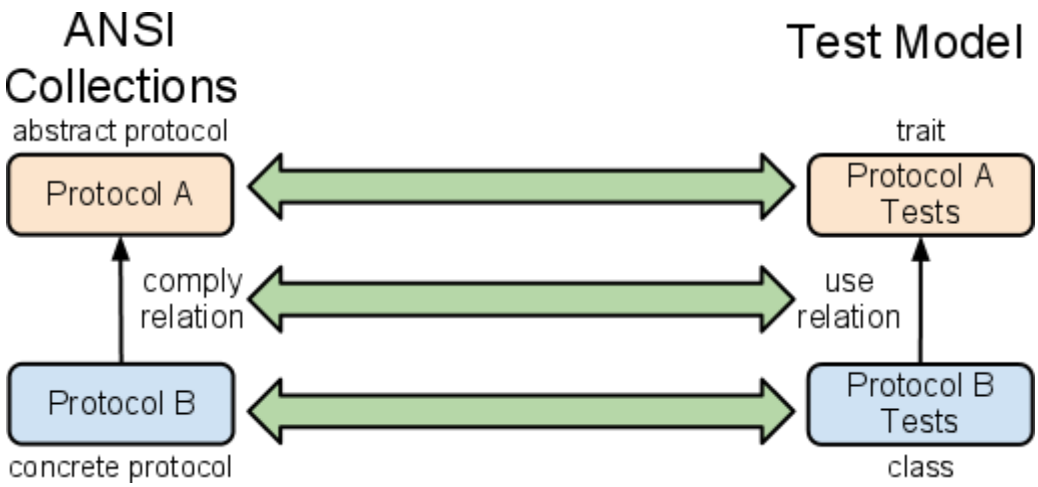

Figura 4 - Relación entre el dominio del problema y el modelo de test

La Figura 5 muestra un diagrama de clases y traits del modelo resultante.

[4]Los protocolos concretos son aquellos que comienzan con mayúscula, y los abstractos los que comienzan con minúscula.

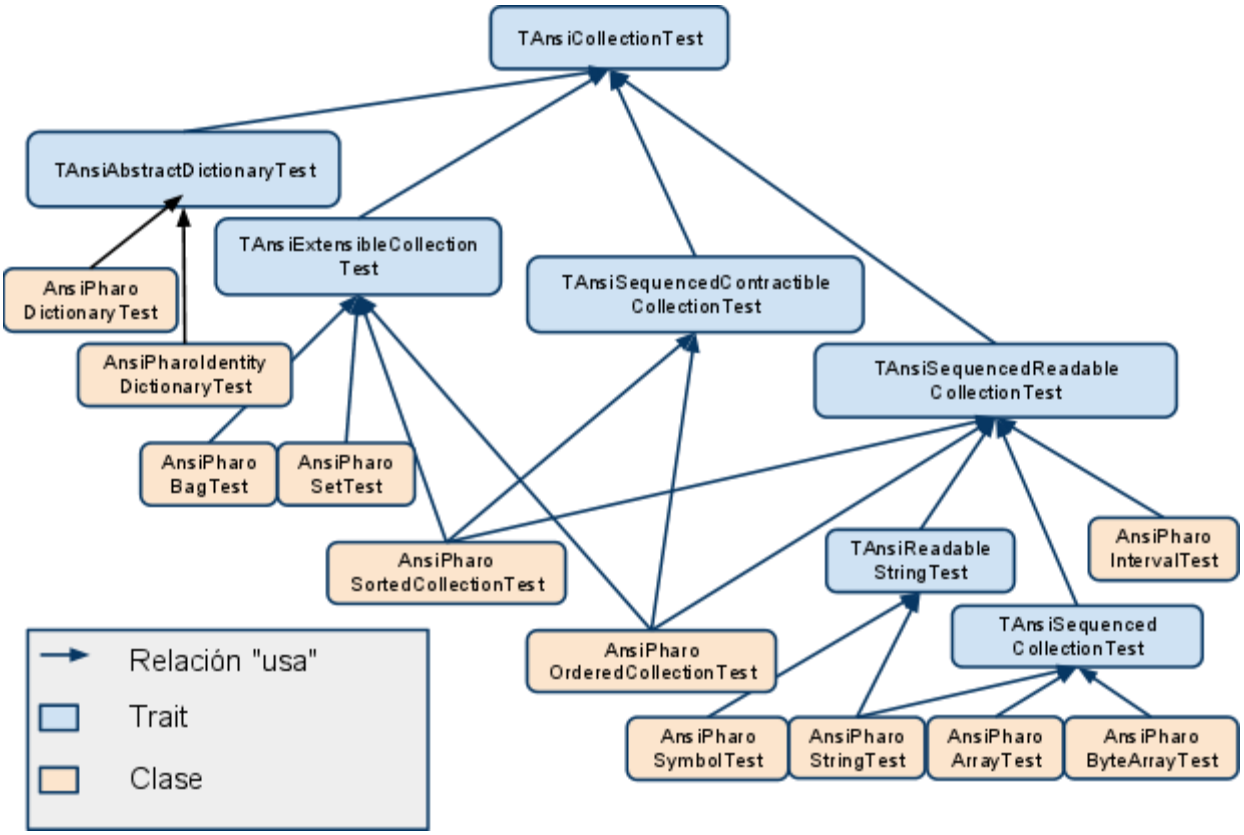

Figura 5 - Diagrama de traits y clases correspondiente al modelo de tests del protocolo ANSI para la jerarquía de Collections de Pharo

De esta manera cada trait contiene los tests correspondientes a un protocolo abstracto, mientras que las clases contienen los tests de los protocolos concretos.
La Figura 6 muestra un ejemplo concreto de la relación entre los protocolos y mensajes del ANSI y el modelo de test.

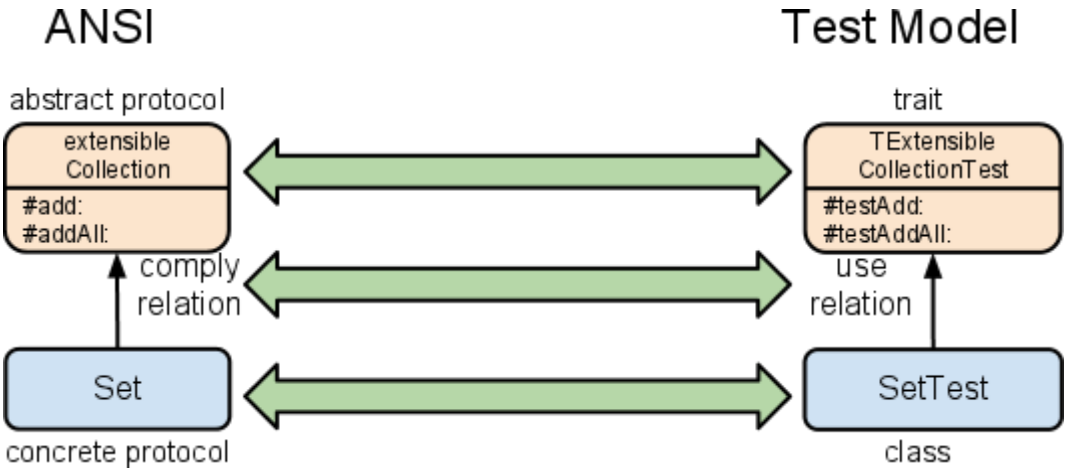

Figura 6 - Ejemplo de relación entre la definición ANSI de Collections y el modelo de test

A su vez en muchos casos los traits que representan los tests de un protocolo abstracto utilizan los traits que ya estaban implementados en Pharo para testear la jerarquía de Collection.
Al momento de utilizar un trait existente, puede que se necesite excluir algún test que no forme parte del protocolo. Los traits permiten excluir métodos al momento de la composición, por lo que este caso es contemplado. La Figura 7 muestra un ejemplo de lo anteriormente explicado.

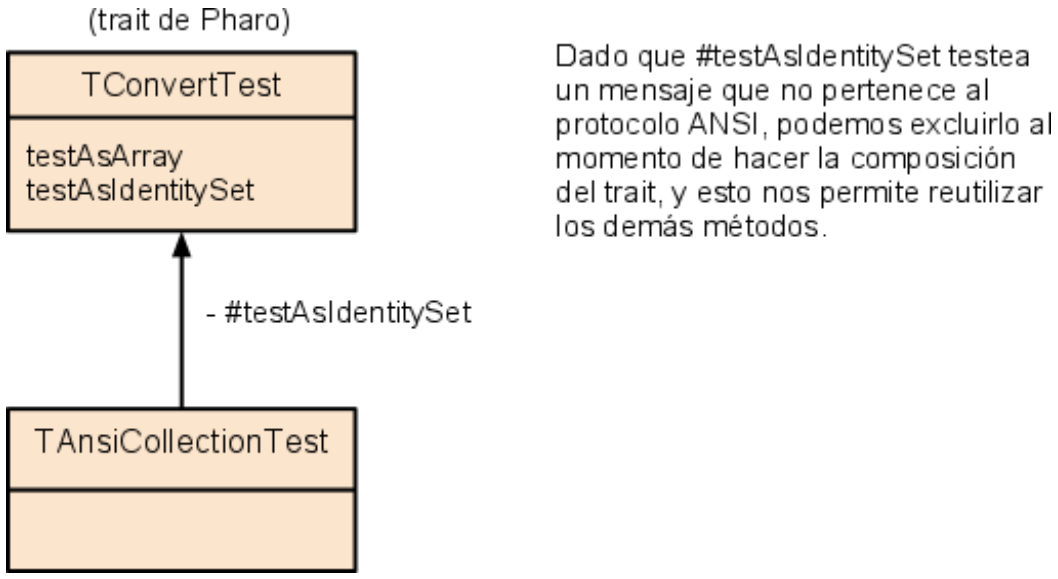

Figura 7 - Ejemplo de exclusión de métodos que no corresponden al ANSI

También en muchos casos existen tests sobre mensajes del protocolo pero que hacen uso de mensajes fuera del protocolo. En estos casos la solución es redefinir el método en el trait local, haciendo que la implementación del mismo utilice solo mensajes del protocolo. Por lo tanto la posibilidad de sobrescritura que tienen los traits ayuda a la reutilización. La Figura 8 muestra un ejemplo de lo anteriormente explicado.

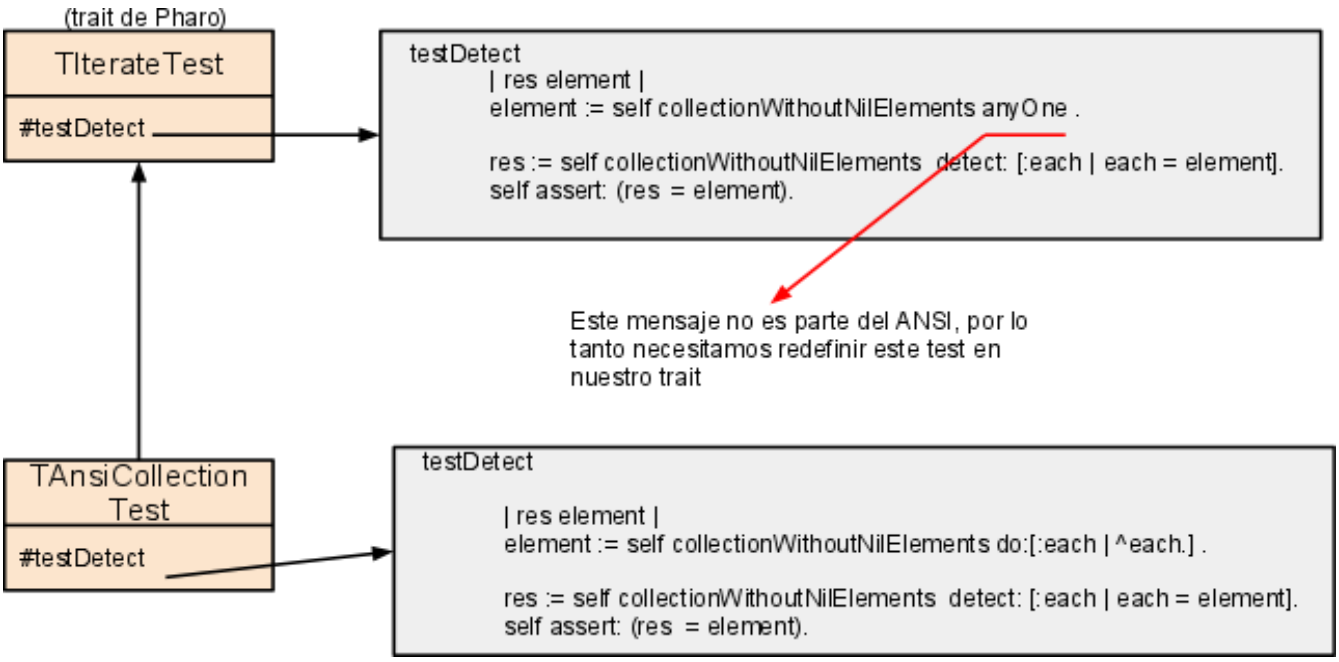

Figura 8 - Ejemplo de Sobrescritura de tests que utilizan mensajes que no son del ANSI

Como se mencionó previamente, la idea es poder reutilizar los test tanto para el modelo actual de Collection, como para el modelo nuevo. Por este motivo los traits son parametrizables a partir de sus requerimientos.
Los tests de protocolos abstractos requieren que se implementen métodos que provean objetos concretos que se utilizarán en los tests. Por lo tanto las clases concretas proveen la implementación de estos requerimientos, y representan los tests de una clase concreta de la jerarquía. Los objetos que proveen estos métodos deben ser instancia de la clase correspondiente al protocolo que se está testeando.
Teniendo en cuenta esto, es posible reutilizar todos los traits correspondientes a los protocolos abstractos, para crear los tests del nuevo modelo de Collection.
La Figura 9 muestra un ejemplo de lo explicado anteriormente.

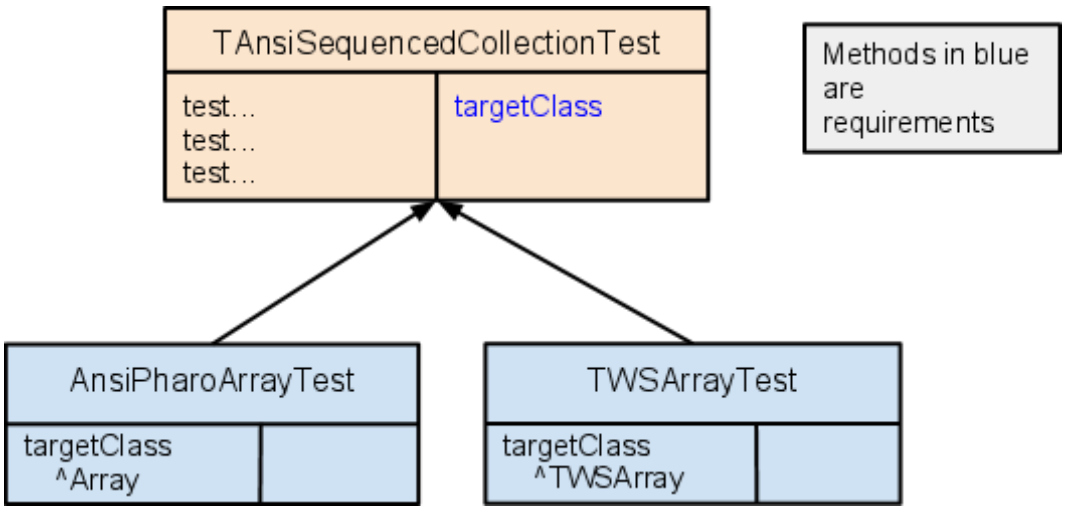

Figura 9 - Ejemplo de reutilización de tests entre modelos parametrizando la clase

4.1.1.1 Modelo de Tests para el Protocolo de Creación de Instancias

El ANSI define tanto el protocolo de Collection para instancias como para la creación de las mismas. Un diagrama de este protocolo se muestra en la sección: Relaciones entre protocolos de Collection del ANSI
Pharo no cuenta prácticamente con ningún test sobre los métodos de creación de instancias. Por lo tanto, se crean traits con tests para los protocolos. El modelo utilizado es análogo al modelo utilizado para el protocolo de instancias de Collection.
Cada trait representa un protocolo de creación de instancias, salvo en el caso de los protocolos *IdentityDictionary Factory* y *Dictionary Factory*, ya que son iguales y se pueden modelar como uno solo.
Cada protocolo es parametrizado mediante un requerimiento, este requerimiento es un método que debe ser implementado devolviendo la clase a testear.
Cabe aclarar que el mensaje `#withAll:` que según el ANSI pertenece a los protocolos: *Dictionary Factory*, *IdentityDictionary Factory* e *initializableCollection Factory* fue movido al protocolo *collection Factory* ya que pertenecía a todos sus "subprotocolos" y con la misma semántica. Además el mensaje #withAll se encuentra implementado en la clase Collection.
Estos traits de creación de instancias se utilizan en las clases que testean los protocolos instancias correspondientes.
La Figura 10 muestra el modelo utilizado.

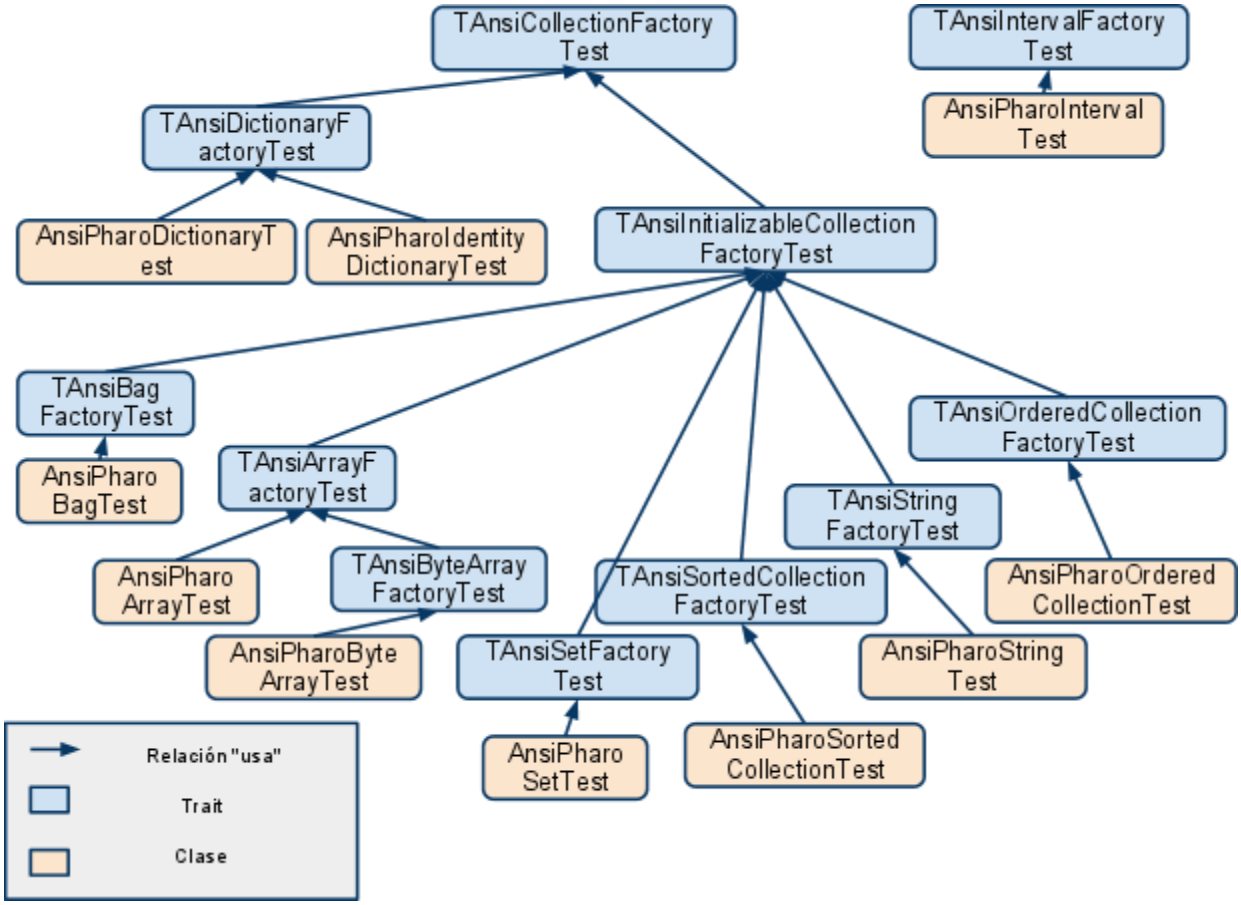

Figura 10 - Modelo de test para los protocolos de creación de instancias del ANSI

4.1.1.2 Modelado de Refinamientos

El ANSI define refinamientos de mensajes de protocolos. Cuando un protocolo, cumple con otro, quiere decir que cumple con todas las definiciones de sus mensajes. En algunos casos, un protocolo puede cumplir con otro, pero puede requerir redefinir cierto comportamiento de mensaje. En estos casos se define un refinamiento de un mensaje.
Un ejemplo de este caso es el mensaje `#new:` definido en el protocolo `CollectionFactory` sin especificar el efecto del parámetro del método. El efecto del parámetro se especifica en los refinamientos del mensaje, y en algunos de estos indica la cantidad de elementos de la colección, mientras que en otros simplemente es una sugerencia al diseñador de la cantidad de elementos que puede contener.
En estos casos, el test del mensaje utiliza un requerimiento para representar los posibles refinamientos. De esta manera los traits que lo utilicen deberán implementar el requerimiento, y así definir el refinamiento del test, sin la necesidad de reimplementar el test completamente y así duplicar código.
A continuación se muestra un ejemplo para el mensaje `#with:with:`

```
TAnsiInitializableCollectionFactoryTest>>testWithWith

	| aCol collection element1 element2 |

	collection := self collectionMoreThan5Elements asOrderedCollection
	copyFrom: 1 to: 2.
	element1 := collection at: 1.
	element2 := collection at: 2.

	aCol := self collectionClass with: element1  with: element2 .
	self assert: (aCol occurrencesOf: element1 ) == ( collection
	occurrencesOf: element1).
	self assert: (aCol occurrencesOf: element2 ) == ( collection
	occurrencesOf: element2).
	self assert: (self additionalConditionForWithTestWithArgs:
	{element1. element2} createdCollection: aCol).


TAnsiArrayFactoryTest>>additionalConditionForWithTestWithArgs: argsCollection
createdCollection: collection
"return a boolean which is the result of an additional condition for the
tests of the methods #with: #with:with: and so on..."
	| index |
	index := 0.
	^collection allSatisfy:
		[:each |
		index := index + 1.
		^each = (argsCollection at: index).
		].
```

El ejemplo muestra cómo TAnsiArrayFactoryTest agrega una condición, que modela el refinamiento, al test `#testWithWith` implementando el mensaje `#additionalConditionForWithTestWithArgs:createdCollection:`.
La condición que se agrega es que los parámetros pasados en el mensaje `#with:with:` se encuentren en el mismo orden en la colección creada.

4.1.2 Comparación con Modelos que Utilizan Subclasificación

Dado que Pharo no cuenta con un modelo de tests de Collection que utilice únicamente subclasificación, sino que es un híbrido que combina subclasificación con traits, debemos buscar otras implementaciones o suponer otros modelos posibles.

En Squeak[7] el modelo de tests no utiliza traits, pero tampoco utiliza subclasificación. Tenemos una clase de test por cada clase de la jerarquía de Collection. Este no es un buen modelo ya que no reutiliza tests que podrían ser compartidos por las clases de Collection.

Supondremos entonces un modelo de tests utilizando subclasificación. Una posibilidad sería tener una clase de test por cada case de la jerarquía a testear, y además que las clases de test mantengan la misma estructura jerárquica que están testeando.
Tomamos entonces como jerarquía de Collection la implementación de Pharo.

Si replicamos esta jerarquía en los tests surgen algunos problemas, como por ejemplo con el mensaje `#add:` .Este mensaje no lo deben responder todos los objetos de la jerarquía. Por ejemplo un array, no debería saber responder el mensaje `#add:` sin embargo en la implementación de Pharo, este mensaje se implementa como `#subclassResponsibility` en la clase Collection, y ArrayedCollection lo implementa como `#shouldNotImplement`.
Entonces debemos analizar dónde se implementa el mensaje `#add:`.
Si es implementado en CollectionTest, todos los objetos de la jerarquía deberían saber responder el mensaje, pero esto no es correcto. Cada subclase que no corresponde que implemente el test de `#add:` puede sobrescribir el método, pero no sería una buena solución, ya que es costoso, además de ser propenso a errores y confuso.
Otra posibilidad es realizar el test OrderedCollectionTest, pero ahora debemos repetir el mismo para las clases BagTest y SetTest ya que se encuentran en ramas distintas de la jerarquía. Repetir el test es una solución que es propensa a inconsistencias, poco mantenible y más costosa a la hora de implementarla.
Si bien el comportamiento del mensaje `#add:` cambia de acuerdo a la clase que lo implementa, existe un comportamiento compartido por todas las implementaciones. Esto es: luego de que se envía el mensaje `#add:`, el objeto pasado como parámetro está incluido en la colección receptora del mensaje.
Por lo tanto este comportamiento compartido debería estar reflejado en un test común a todas las implementaciones. Este test debería poder escribirse una vez y ser reutilizado sobre todas las implementaciones del mensaje `#add:` y solamente en ellas. Sin embargo por lo explicado anteriormente, esto no es posible en el modelo.

Este problema surge porque los métodos que necesitan ser compartidos cumplen con una característica que hace que sea imposible representarlos correctamente utilizando subclasificación simple.
Se puede encontrar más detalle con respecto a esto en la sección Métodos Sobrantes.

Este tipo de problemas, no surgieron utilizando el modelo de traits-sin-subclasificación, simplemente se crea el test del mensaje en un trait, parametrizándolo para diferentes clases y se utiliza en cada clase que deba implementarlo.

4.2 Nuevo Modelo de Collection con Traits sin Subclasificación

Con el desarrollo del modelo de test terminado podemos comenzar con la creación del nuevo modelo de Collections y siguiendo la metodología descripta anteriormente.
Esta sección explicará, en detalle, los pasos seguidos durante el desarrollo del nuevo modelo. Se explicará en desarrollo del modelo en el orden en el que fue evolucionando. La descripción se realiza clase por clase. Para cada una de estas se detallan los cambios realizados al modelo, las métricas que corresponden al desarrollo de cada una, y detalles relevantes al desarrollo.

4.2.1 Definición de Métricas

Se definen las siguientes métricas para evaluar el desarrollo:

- **Métrica de Desarrollo**: Indica la cantidad de tests pasados sobre la cantidad de métodos implementados/movidos. Esto se debe a que un método puede ser implementado directamente en la clase, o puede ser movido desde otra clase hacia un trait de manera tal que sea utilizado por la clase que se está desarrollando.

- **Métodos según Procedencia**: Divide los métodos de la clase desarrollada en 3 categorías:

    - **Métodos Movidos**: son aquellos que se implementaron en la clase a partir de haberlos movido de otra hacia un trait compartido por ambas.

    - **Métodos Implementados**: son aquellos que se implementaron directamente en la clase en cuestión

    - **Métodos utilizados de Traits Inicialmente**: son aquellos que se implementaron en la clase al momento de definirla. En la definición de la clase se especifica qué traits utiliza, y por lo tanto utiliza los métodos de estos.

4.2.2 Evolución del modelo

La metodología de trabajo explicada anteriormente tiene como objetivo el desarrollo de un modelo en forma evolutiva. La idea es primero desarrollar de la manera más simple posible con el objetivo de cumplir un test y luego realizar abstracciones a medida que son necesarias, y no antes. Esto significa que cada refactoring realizado al modelo hace que el mismo evolucione iterativamente asegurándonos que sigue cumpliendo con los tests.
A continuación, se describirán detalles del desarrollo de este modelo, indicando las decisiones de diseño y los resultados parciales que se van obteniendo. También se muestran métricas que tienen como objetivo reflejar los esfuerzos de desarrollo del trabajo realizado.

4.2.2.1 El Comienzo: implementando TWSArray

La primera clase a implementar fue TWSArray. El motivo de esta elección es que no depende de otras clases, y además muchas de estas otras dependen de ella. La implementación de esta clase implica aplanar los métodos de Array en TWSArray. Aplanar significa tener todos los métodos que están en la rama de la jerarquía donde se encuentra la clase Array en una sola clase TWSArray. Además es necesario eliminar la utilización de *super* de los métodos que corresponda.
No es posible implementar completamente TWSArray, ya que los métodos de conversión como `#asSet` o `#asOrderedCollection` no pueden escribirse porque todavía no se crearon las clases que representan los comportamientos de un Set o a una OrderedCollection.
De momento no se crea ningún trait, ya que es la única clase hasta el momento.

4.2.2.2 Primeros Traits: TWSOrderedCollection

La siguiente clase a implementar es TWSOrderedCollection. El motivo por el cual se elige continuar con esta clase es que sólo depende de TWSArray.
Al momento de implementar esta clase encontramos métodos que tienen el mismo comportamiento que el definido por TWSArray. Ante estos casos es necesario compartir comportamiento entre las instancias de estas clases. El comportamiento es compartido a través de traits.
En principio la metodología para definir un método en un trait es: identificar primero a qué protocolo pertenece el mensaje que el método implementa. Luego se crea un trait que corresponde al protocolo del mensaje, y se implementa el método en el mismo.
En los casos en los que el mensaje tenga refinamientos[5] el método se implementará en el trait correspondiente al protocolo donde se define el refinamiento, pero solo si es necesario compartirlo.
También puede existir el caso en el que un método se encuentre redefinido por una cuestión implementativa y no de comportamiento. En estos casos en primera instancia se intentarán implementar en alguno de los traits de protocolo, hasta que la evolución del modelo requiera de otro tipo de solución. La implementación del método debe respetar el protocolo del trait al cual se mueva.
Como resultado de la implementación de la clase TWSOrderedCollection, y siguiendo con las pautas anteriormente explicadas, se crearon los traits: TCollection, TSequencedReadableCollection y TSequencedCollection para compartir comportamiento con la clase TWSArray.
La Figura 11 muestra la evolución del modelo hasta este punto.

[5] En algunos casos, un protocolo puede cumplir con otro, pero puede requerir redefinir cierto comportamiento de mensaje. En estos casos se define un refinamiento de un mensaje.

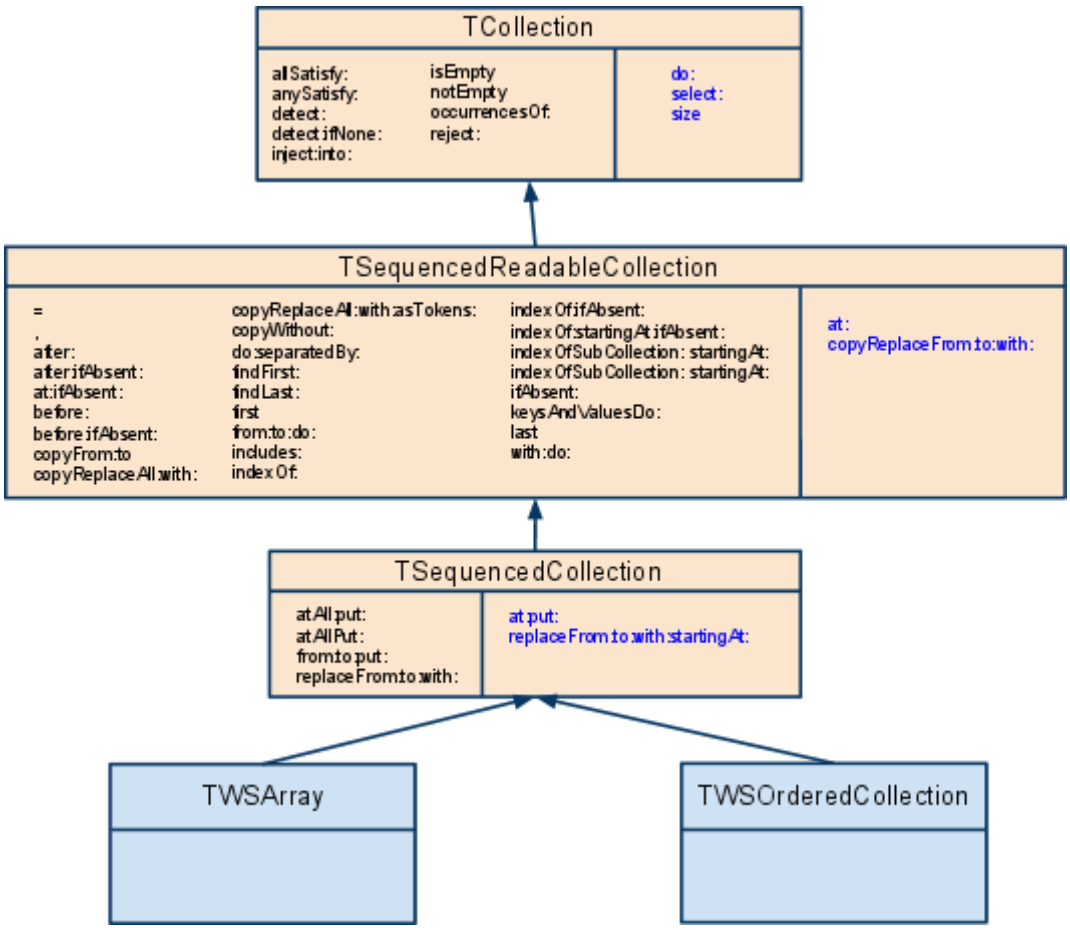

Figura 11 - Diagrama de Clases + Traits mostrando cómo evoluciona el modelo y de qué manera se comparte comportamiento entre TWSArray y OrderedCollection

Como se ve en la Figura 11, se comparten 39 métodos a través de los traits. Resumiendo, el procedimiento que se utilizó para llegar a este modelo fue:

- Crear un trait por cada protocolo compartido entre las clases
- Ante cada mensaje para el cual comparten la implementación se crea un método en el trait correspondiente al protocolo del mensaje, de modo que lo compartan.
- El método compartido se implementa en un trait que represente un protocolo que cumpla con el mensaje que el método implementa.
- El mensaje correspondiente al método compartido debe pertenecer al protocolo que el trait, donde fue implementado el método, representa.

4.2.2.3 Reutilización de Traits: TWSByteArray

Se continúa la evolución del modelo implementando la clase TWSByteArray. La razón por la cual continuamos con esta clase, es que comparte el protocolo con TWSArray.
En este caso se comenzaron a compartir métodos de clase a través de traits, lo cual resulta simple desde el punto de vista del desarrollador, ya que cuando una clase usa un trait automáticamente su metaclase usa la correspondiente *trait class*, de modo que los métodos definidos en la *trait class* serán utilizados por la metaclase.
TWSByteArray comparte el 98% de sus métodos con TWSArray. La metodología utilizada para compartir los métodos entre TWSArray y TWSByteArray es: si existe un método que TWSByteArray requiere y su implementación está en TWSArray, el método se mueve al trait TSequencedCollection. Esta refactorización no debería afectar a TWSOrderedCollection ya que si se utilizara otra implementación para el método movido, ésta debería estar definida en la clase, de forma que sobrescribe los métodos utilizados por el trait.
No se crearon nuevos traits, simplemente se movieron métodos de la clase TWSArray al trait TSequencedCollection.

4.2.2.3.1 Métricas de Desarrollo

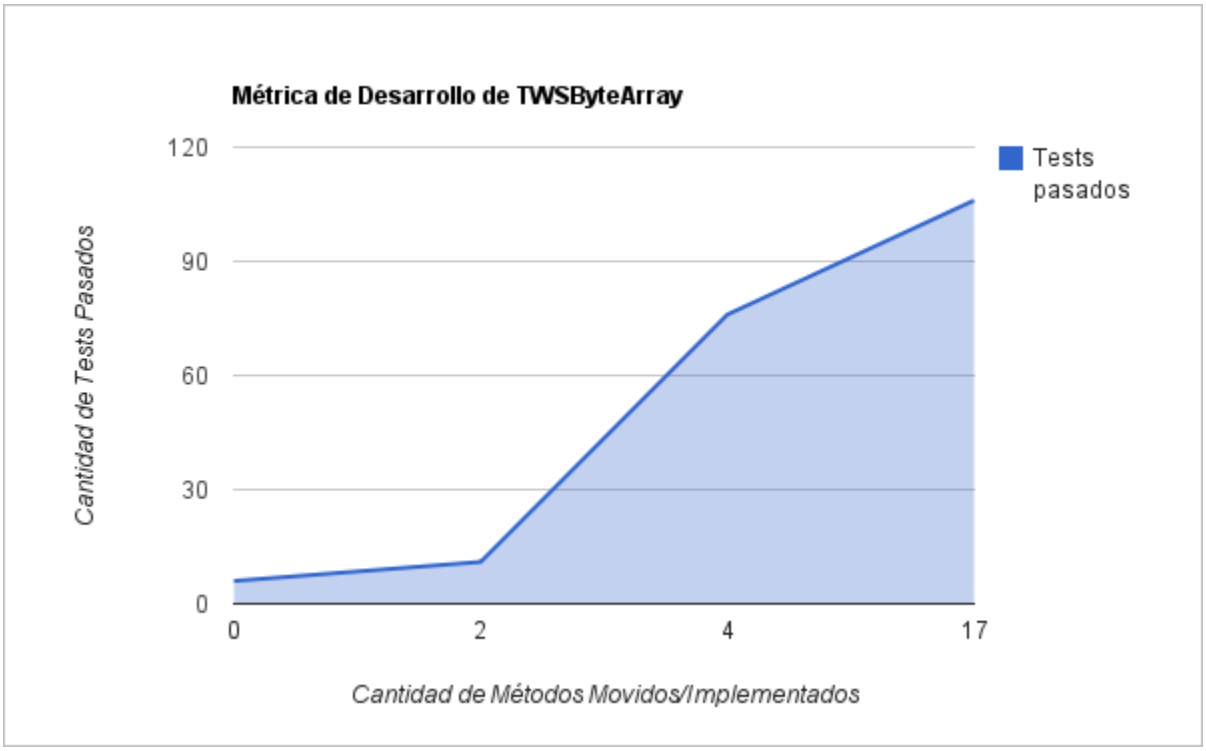

Figura 12 - Métrica de Desarrollo de TWSByteArray

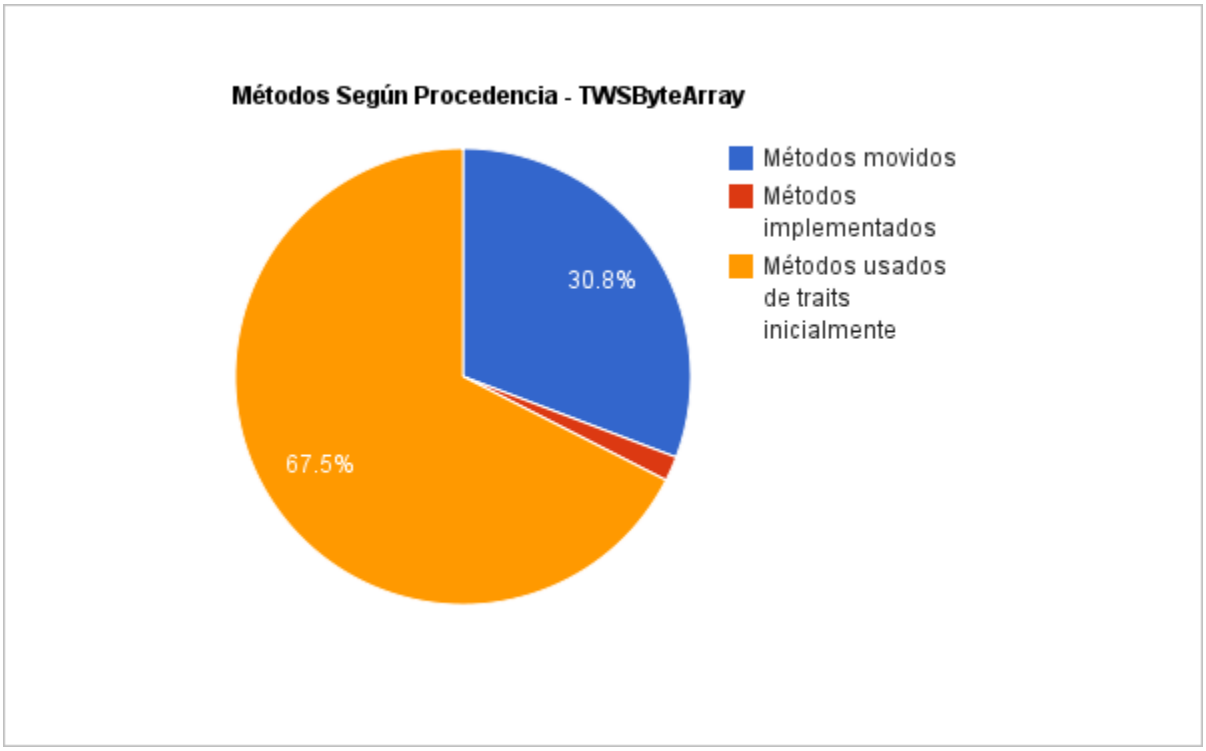

Figura 13 - Métrica de Métodos Según Procedencia de TWSByteArray

Las figuras 12 y 13, muestran el esfuerzo del desarrollo de la clase TWSByteArray. Podemos observar que el esfuerzo es muy bajo, ya que sólo fue necesario implementar en la clase el 1,7% de los métodos, mientras que el resto fue reutilizado.

4.2.2.4 Cambio de rumbo: TWSSortedCollection

La clase TWSSortedCollection implementa el protocolo de SortedCollection, el cual comparte 58 mensajes con el protocolo OrderedCollection por lo tanto se elige para continuar la evolución del modelo y reutilizar lo más posible el comportamiento de OrderedCollection.

4.2.2.4.1 Modificación del Modelo

Al momento de comenzar la implementación de la clase TWSSortedCollection necesitamos compartir comportamiento con la clase TWSOrderedCollection.
Los traits por los cuales estas dos clases comparten comportamiento son: TCollection y TSequencedReadableCollection.

Concretamente ambas clases comparten la implementación del método de clase `#new:`. Pero si este mensaje se moviera a alguno de los traits que comparten, entonces la clase TWSByteArray cambiaría su implementación del mensaje `#new:` ya que utiliza la implementación de la clase Behavior.

Luego tenemos que la implementación de `#new` en TWSArray sería equivalente a la encontrada en la clase Behavior, entonces puede compartir el mensaje `#new:` de TWSArray con TWSByteArray a través del trait TSequencedCollection. Pero esto tampoco soluciona el problema, ya que si se implementa el método `#new:` de TWSOrderedCollection en TSequencedReadableCollection, el mismo es sobrescrito por TSequencedCollection y, por lo tanto, TWSOrderedCollection no tendría la implementación correcta.
Además del caso del mensaje de clase `#new:` existen otros casos, como por ejemplo el mensaje `#copyFrom:to:` tiene una implementación en TWSArray y TWSByteArray y una segunda en TWSOrderedCollection y TWSSortedCollection. Este mensaje se encuentra definido en el protocolo SequencedReadableCollection.
Con esto queremos mostrar que solamente podemos crear un nuevo trait.
Dada esta conclusión queda en evidencia que el modelo anteriormente planteado requiere de ciertas modificaciones. En principio no es posible mantener un único trait por protocolo y esto no estaba del todo contemplado en el diseño planteado cuando se implementó TWSOrderedCollection.

Lo que se puede deducir de este problema es que no se están representando protocolos sino implementaciones de los mismos y que, para un mismo protocolo, pueden existir diferentes implementaciones. Por este motivo no es posible mantener un trait por protocolo. Pero, por otro lado, queremos mantener la idea de tener los métodos de un protocolo agrupados en un trait, de esta manera se evita que se remuevan métodos de un trait al momento de componerlo.
Dicho esto, el nuevo modelo ahora tendrá un trait por cada implementación de un protocolo, y así las clases podrán utilizar la implementación que necesiten del mismo. Los métodos que sean comunes entre diferentes implementaciones de protocolos se compartirán a través de traits.
Para ilustrar la solución anteriormente explicada, se muestra a continuación un diagrama traits y clases.

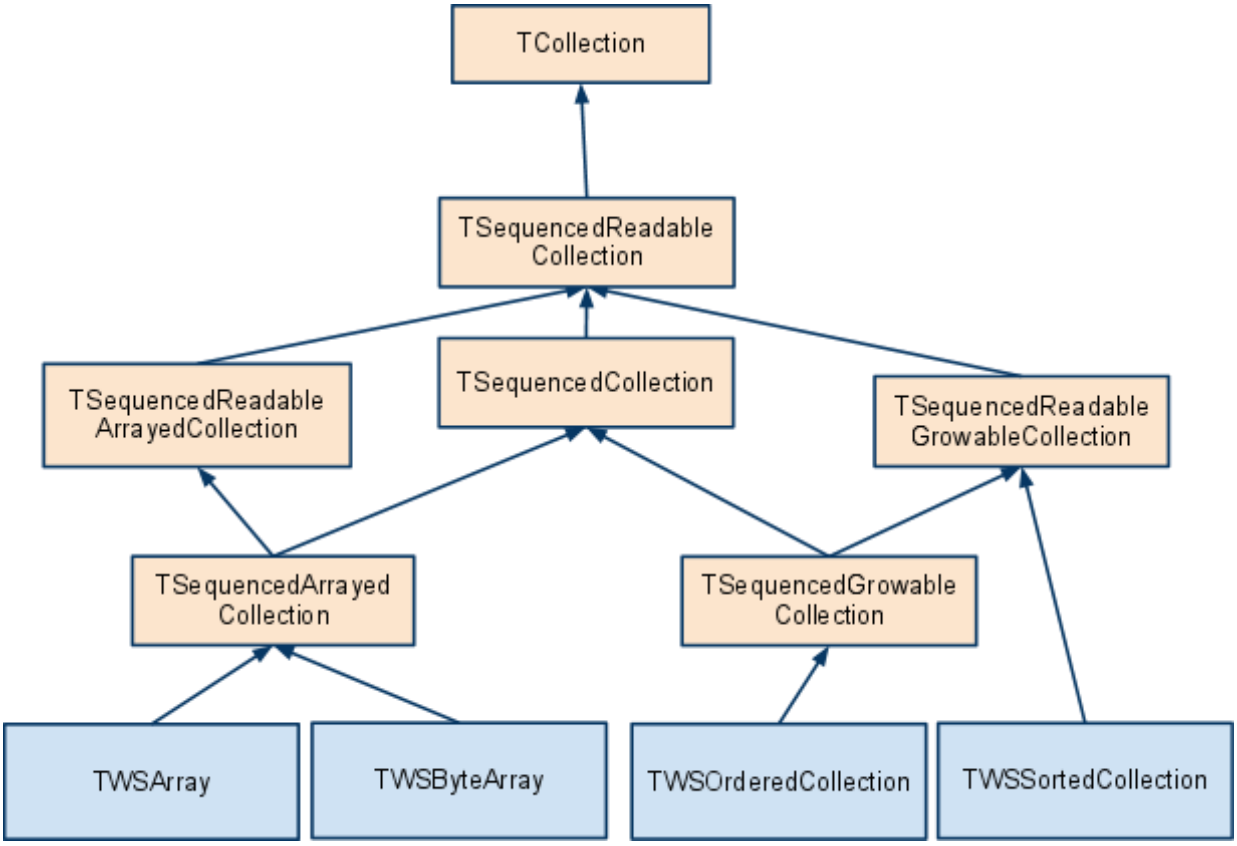

Figura 14 - Diagrama de traits y clases del nuevo modelo modificado

La Figura 14 muestra 2 implementaciones del protocolo SequencedReadableCollection, una de ellas es la versión arrayed y la otra growable. Justamente se eligieron estos nombres porque la mayoría de las diferencias en la implementación surgen porque en algunos casos se necesita manejar el crecimiento de la colección como es el caso de las growable, y en otras no, como es el caso de las arrayed.
Debemos aclarar que una colección es growable si su tamaño puede aumentar. Si la misma tiene un tamaño fijo es arrayed.
Dado que existen métodos que son compartidos por ambas implementaciones, los mismos los definimos en un trait aparte: TSequencedReadableCollection, el cual ambas implementaciones utilizan.
Por otro lado queremos seguir manteniendo la relación entre protocolos. Esto es: si un protocolo cumple con otro, entonces en nuestro modelo queremos que esto quede representado. Por este motivo es que, por ejemplo, ambas implementaciones del protocolo SequencedReadableCollection utilizan el trait TCollection.
En los casos en que existan varias implementaciones de un protocolo A con el que otro B debe cumplir, se debe utilizar la implementación de A que es compatible con la implementación de B. Este es el caso de las implementaciones del protocolo SequencedCollection, donde cada implementación utiliza el trait de la implementación del protocolo SequencedReadableCollection que es consistente con el. Por ejemplo: TSequencedArrayedCollection utiliza el trait TSequencedReadableArrayedCollection, y no debería utilizar la implementación TSequencedReadableGrowableCollection.

4.2.2.4.2 Implementando el Cambio de Modelo

Una vez definido el nuevo diseño comenzamos la implementación del mismo.
Se crean los 4 nuevos traits sin métodos: TSequencedReadableArrayedCollection, TSequencedReadableGrowableCollection, TSequencedArrayedCollection, TSequencedGrowableCollection. Luego se modifican las relaciones de uso de traits y se corren todos los tests verificando que siguen corriendo exitosamente.

Dado que SortedCollection hereda de OrderedCollection en el modelo actual de Collections de Pharo, se comparten muchos métodos que acceden a variables de instancia directamente. Estos métodos no es posible compartirlos utilizando traits, por lo menos no sin hacer un refactoring previo.
Para solucionar esto y poder compartir este comportamiento, los métodos deben ser refactorizados para que no accedan directamente a las variables de instancia. En vez de accederlas directamente los métodos utilizarán métodos de acceso o *accessors*, también conocidos como *getters* y *setters*.

Al momento de implementar mensajes como `#remove:` en TWSSortedCollection encontramos que es necesario reutilizar la implementación de TWSOrderedCollection. Según el modelo actual deberíamos hacerlo en el trait TSequencedReadableGrowableCollection, sin embargo el mensaje `#remove:` no pertenece al protocolo SequencedReadableCollection, por lo tanto no sería correcto implementarlo ahí. El mensaje `#remove:` pertenece al protocolo ExtensibleCollection, este protocolo no tiene un trait que represente su implementación. Entonces creamos un nuevo trait TExtensibleCollection, hacemos que TWSOrderedCollection y TWSSortedCollection lo utilicen, y movemos los métodos compartidos que corresponden al protocolo.

De manera análoga a cómo surgió el trait TExtensibleCollection fue necesario crear TSequencedContractibleCollection para compartir métodos correspondientes a mensajes del protocolo SequencedContractibleCollection.

En el modelo actual de Collections de Pharo, la clase SortedCollection hereda el mensaje `#addLast:` de OrderedCollection. El mensaje `#addLast:` pertenece al protocolo de OrderedCollection, pero no pertenece al protocolo de SortedCollection, lo cual tiene sentido ya que no se debería poder controlar dónde de agrega un elemento en SortedCollection, ya que la clase misma es responsable de conocer dónde agregar un nuevo elemento.
Sin embargo el mensaje `#addLast:` se utiliza en SortedCollection en ciertos casos por una cuestión de performance, lo cuál es válido. El método `#addLast:` no realiza un control para dejar la estructura interna ordenada, por lo tanto es más *performante* pero debe ser utilizado con cuidado, y solamente debe ser utilizado internamente.

Es por esto que en el nuevo modelo queremos obtener una mejor representación de este caso definiendo un mensaje privado[6] `#internalAddLast:`. El cual tiene el mismo comportamiento que el mensaje `#addLast:` y es compartido por las clases TWSSortedCollection y TWSOrderedCollection a través del trait TExtensibleCollection. Y en TWSOrderedCollection definimos un mensaje `#addLast:` que simplemente reutiliza el método privado.

Como resultado de la implementación de TWSSortedCollection la Figura 15 presenta el modelo resultante mediante un diagrama de traits y clases.

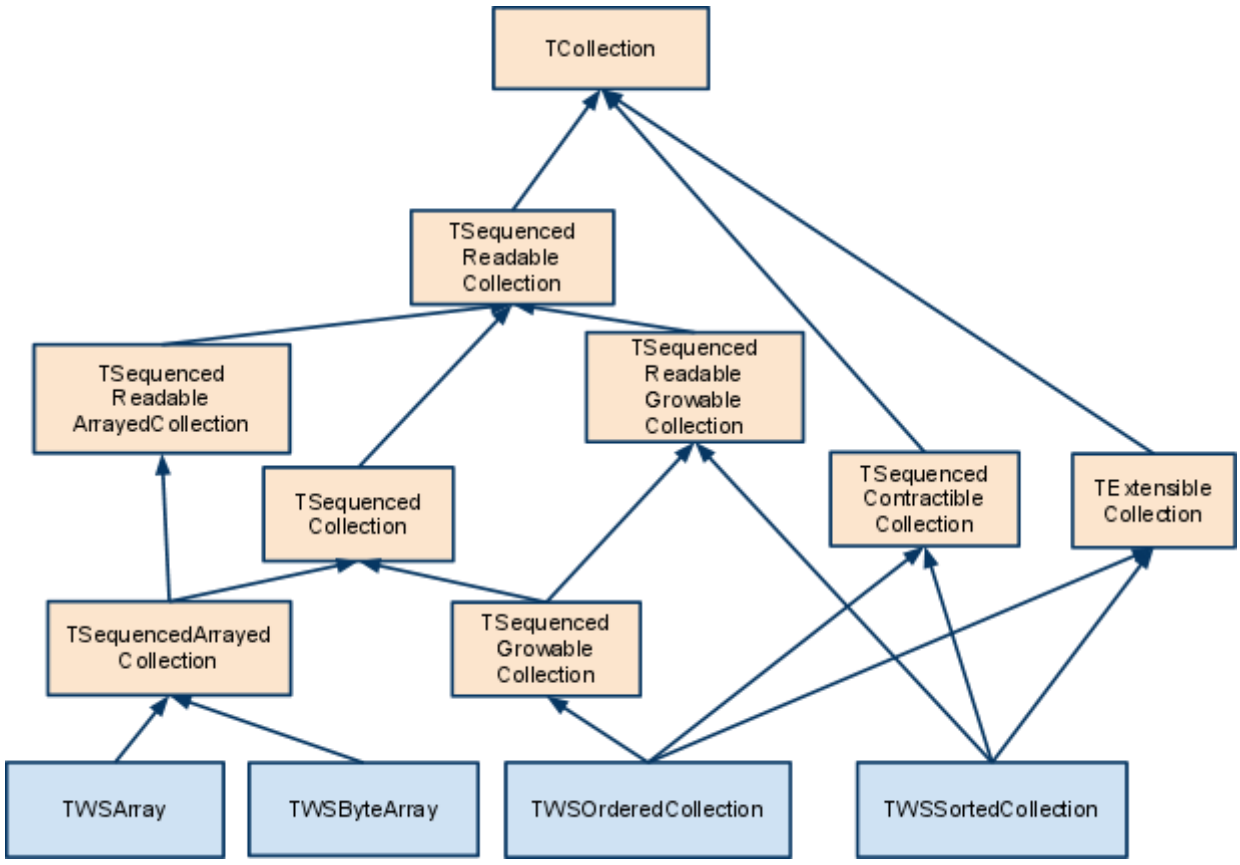

Figura 15 - Diagrama de traits y clases del nuevo modelo luego de la implementación de la clase TWSSortedCollection

4.2.2.4.3 Métricas de Desarrollo

En esta sección presentamos métricas que surgieron del desarrollo de la clase TWSSortedCollection.

[6] En Smalltalk no existe la posibilidad de impedir que se envíen mensajes privados a una clase, la forma utilizada es definirlos dentro de un protocolo llamado *private*.

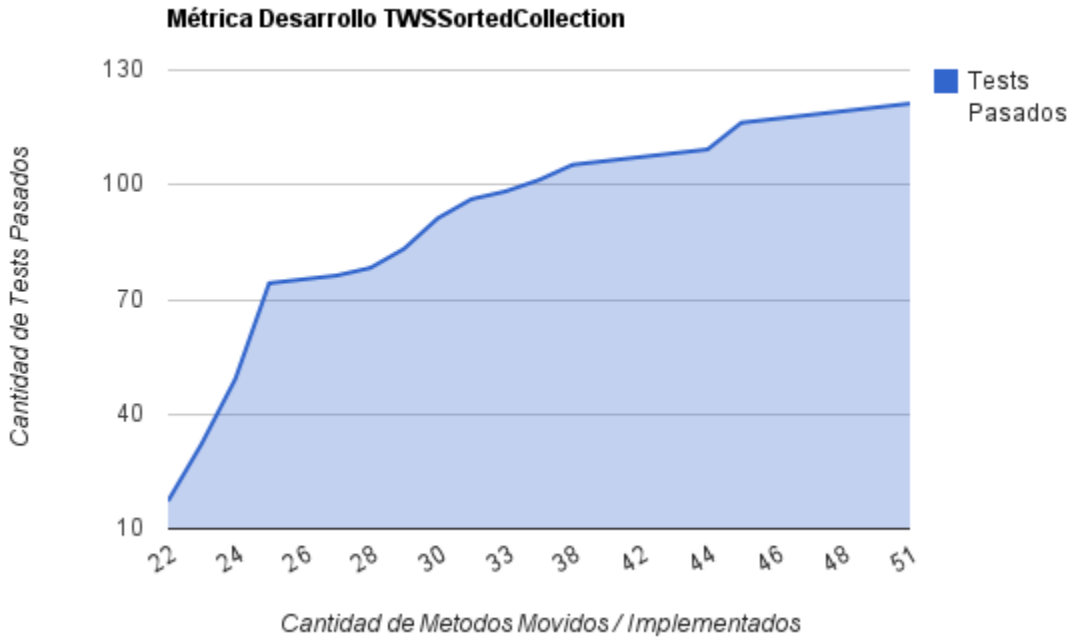

Figura 16 - Métrica de Desarrollo de la clase TWSSortedCollection

El gráfico de la Figura 16 muestra los tests pasados en base a la cantidad de métodos movidos o implementados. Los métodos movidos son aquellos que se movieron de una clase hacia algún trait para que la clase que estamos desarrollando lo utilice. Los métodos implementados son aquellos que son directamente implementados en la clase que estamos desarrollando.

Como se puede observar en el gráfico, en la primera fase del desarrollo es donde se pasan mayor cantidad de tests implementando/moviendo menos métodos. Esta característica está relacionada con las dependencias entre métodos. Por ejemplo, al momento de comenzar el desarrollo, la clase ya cuenta con algunos métodos implementados, producto de utilizar traits, pero los mismos no pasan los tests, porque requieren que otros métodos sean implementados. De esta manera, lo que generalmente pasa, es que cuando implementamos los requerimientos del trait, se pasan los tests de los métodos que implementamos, más los métodos del trait, y eso hace que la curva tenga un mayor ángulo al principio.

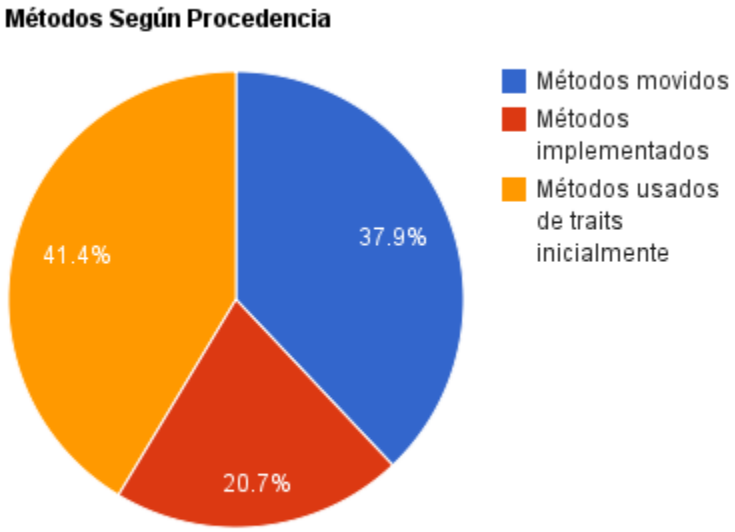

Figura 17 - Métrica de Métodos Según Procedencia de la clase TWSSortedCollection

El gráfico de la Figura 17 muestra el resultado de la implementación de TWSSortedCollection. El mismo divide a los métodos de la clase en 3 grupos y muestra la cantidad de cada grupo con respecto al total.
Se puede ver que la mayoría de los métodos de la clase (41.4%) provienen de traits, pero que ya habían sido movidos antes de comenzar con la implementación de la clase.
Luego con el 37.9% tenemos métodos que provienen de moverlos de otra clase a un trait durante el desarrollo de TWSSortedCollection.
Tan solo el 20.7% son métodos implementados directamente en TWSSortedCollection, los mismos corresponden a métodos que no son reutilizados por ninguna otra clase.
Por lo tanto el 79.3% (la sumatoria de métodos movidos y métodos implementados) de los métodos de la clase son compartidos a través de traits, lo cual reduce el considerablemente el esfuerzo de desarrollo de la clase.

4.2.2.5 TWSSet

Continuamos con la implementación de la clase TWSSet, ya que no depende de clases no implementadas, y puede llegar a reutilizar los métodos del trait TExtensibleCollection.

Al momento de utilizar el trait TExtensibleCollection en TWSSet encontramos un problema: el mismo requiere implementar métodos de acceso `#firstIndex` `#lastIndex` y otros más. TWSSet no debería implementar dichos métodos. Este es el mismo caso que el de

TWSSortedCollection: TWSSet requiere otra implementación del protocolo ExtensibleCollection.
Como solución creamos un nuevo trait TExtensibleOrderedCollection, donde tenemos las implementaciones particulares utilizadas en TWSOrderedCollection y TWSSortedCollection. El trait TExtensibleOrderedCollection utilizará TExtensibleCollection, que contendrá los métodos del protocolo compartidos con TWSSet.
Los métodos del protocolo que sean particulares de la implementación de TWSSet, por el momento se implementan directamente en la clase.
La Figura 18 muestra lo explicado anteriormente mediante un diagrama de traits y clases.

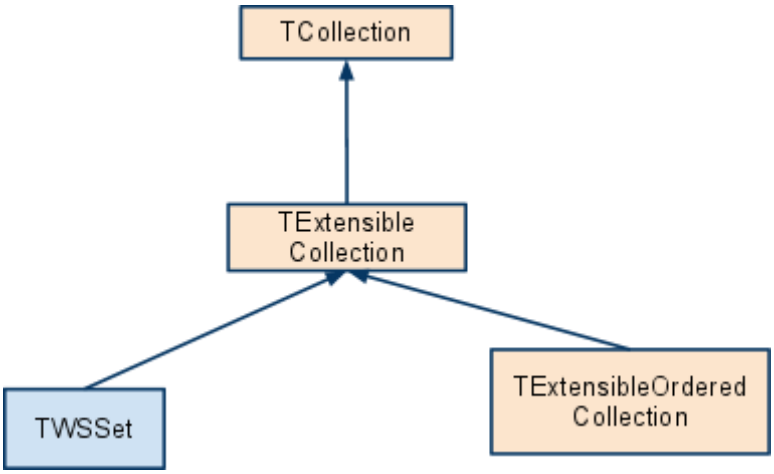

Figura 18 - Diagrama de traits y clases. Resultado de la implementación de TWSSet

4.2.2.5.1 Métricas de Desarrollo

Figura 19 - Métrica de Desarrollo de la clase TWSSet

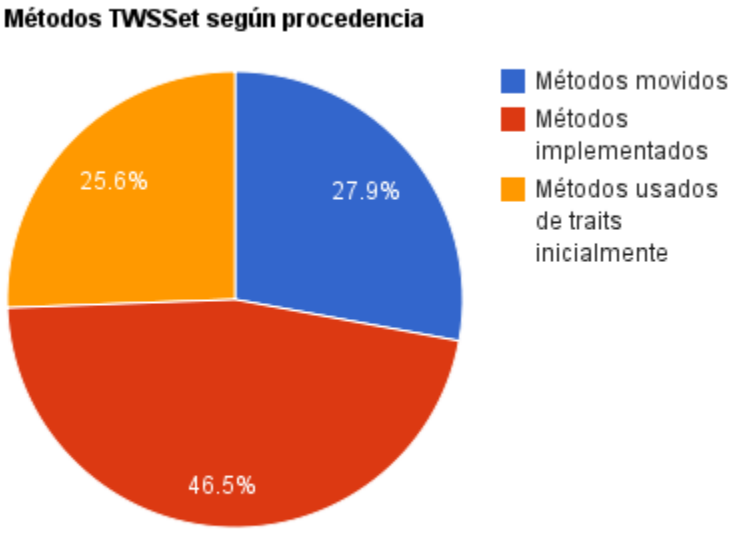

Figura 20 - Métrica de Métodos Según Procedencia de la clase TWSSet

Las figuras 19 y 20, muestran el esfuerzo de desarrollo de la clase TWSSet. Podemos observar que el esfuerzo de desarrollo es mayor que en el caso de TWSByteArray y SortedCollection, ya que se debió implementar el 46.5% de los métodos de la clase. La cantidad de métodos que hay que implementar está relacionada con la cantidad de comportamiento que TWSSet comparte con las clases implementadas hasta el momento. Podemos decir entonces que el esfuerzo de desarrollo de TWSSet es alto comparado con las clases implementadas hasta el momento.

4.2.2.6 TWSDictionary

Continuamos con el desarrollo de la clase TWSDictionary. La misma solo comparte el protocolo de Collection con las demás clases hasta el momento desarrolladas. Sin embargo, la implementación de Dictionary de Pharo comparte mucho comportamiento con la implementación de Set, ya que ambas heredan de HashedCollection. Muchos de estos métodos son privados, y se usan para el manejo de las variables de instancia. Estos métodos acceden directamente a estas variables, por lo tanto para compartirlos, debemos refactorizarlos.
Una vez hecho el refactoring podemos compartirlos mediante un trait. Para esto creamos un nuevo trait llamado THashedCollection. A través del mismo se comparten 6 métodos privados, 1 método del protocolo de Collection, y 2 de CollectionFactory.

Encontramos que TWSDictionary utiliza la misma implementación del método de clase `#withAll:` que se encuentra en TExtensibleCollection. Pero no es posible que TWSDictionary utilice el trait TExtensibleCollection, ya que ExtensibleCollection es un protocolo que TWSDictionary no implementa. Tampoco es posible mover la implementación de `#withAll:` de TExtensibleCollection a THashedCollection, ya que, por ejemplo TWSOrderedCollection utiliza este método y no utiliza THashedCollection. Mover el método a TCollection tampoco sería correcto, ya que de este modo todas las clases utilizarían este método, y por ejemplo, Interval no lo debería implementar.
Podemos ver que la implementación del mensaje `#withAll:` no necesariamente tiene que estar en la implementación de alguno de los protocolos de mensajes de instancia. De hecho el mensaje `#withAll:` no pertenece a ningún protocolo de instancia sino a los protocolos de clase. Sin embargo, parecía conveniente tener la implementación del mensaje de clase `#withAll:` en la implementación del protocolo ExtensibleCollection, ya que el mismo utiliza mensajes de inicialización que pertenecen a ese protocolo. El problema es que el mensaje `#addAll:` no solo pertenece al protocolo ExtensibleCollection, sino que también pertenece al protocolo de AbstractDictionary.
La mejor solución a este problema parece separar la implementación de este mensaje en un nuevo trait TInitializableWithCollectionFactory. Y para hacer explícito el hecho de que la implementación del mensaje `#withAll:` utiliza el método de instancia `#addAll:` agregamos el mismo como un requerimiento del trait.

La Figura 21 muestra el diagrama de clases y traits resultante.

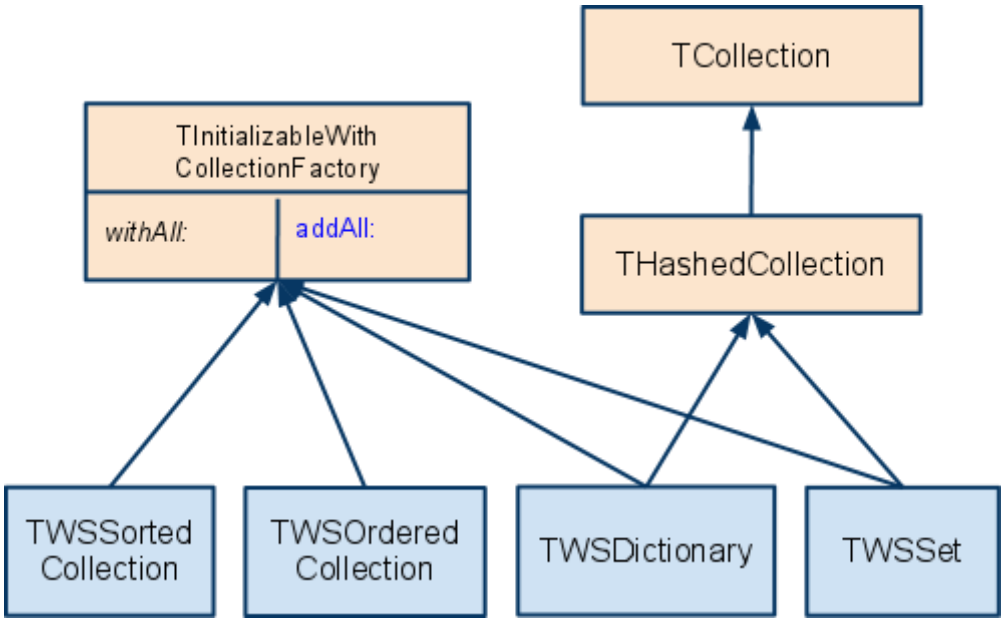

Figura 21 - Diagrama de traits y clases del modelo luego de la implementación de TWSDictionary

4.2.2.6.1 Métricas de Desarrollo

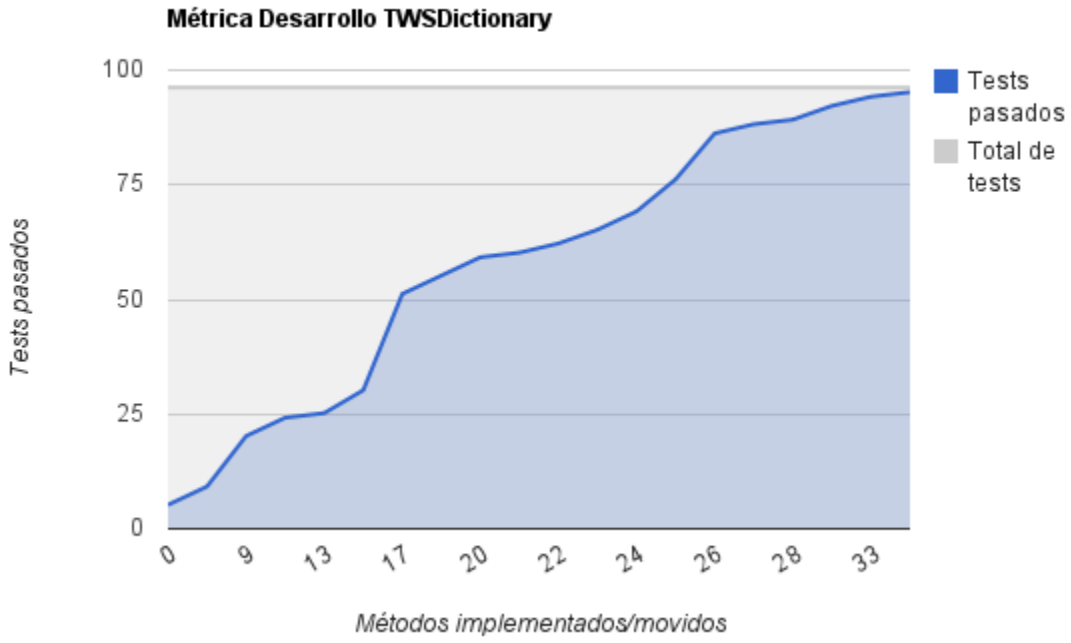

Figura 22 - Métrica de Desarrollo de la clase TWSDictionary

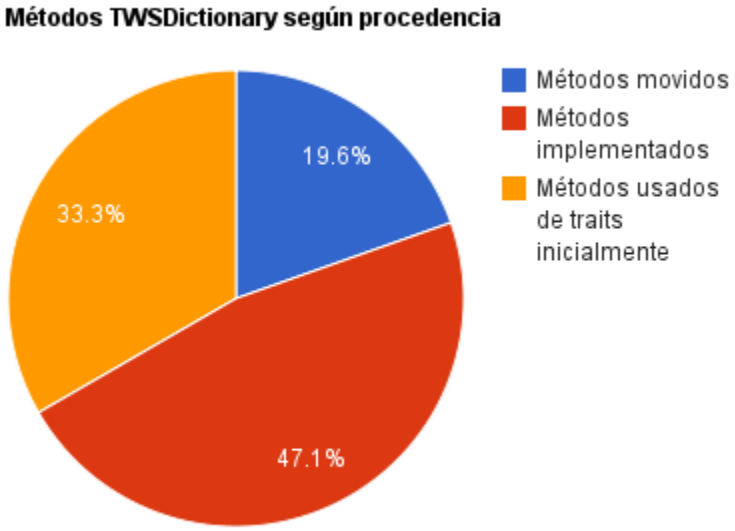

Figura 23 - Métrica de Métodos Según Procedencia de la clase TWSDictionary

Las figuras 22 y 23, muestran el esfuerzo de desarrollo de la clase TWSDictionary. Podemos observar que se implementaron el 47% de los métodos, lo cuál se relaciona con que TWSDictionary sólo comparte el 57% de sus mensajes con el resto de las clases implementadas hasta el momento. Del 57% de mensajes compartidos, se pudo compartir el 52,9% de métodos que los implementan, lo cual indica alto porcentaje de métodos compartidos.

4.2.2.7 TWSIdentityDictionary

Continuamos con el desarrollo de la clase TWSIdentityDictionary, ya que la misma comparte todos los mensajes con TWSDictionary.
Dado que estamos creando abstracciones a medida que las necesitamos, todavía no tenemos un trait para compartir comportamiento entre TWSDictionary y TWSIdentityDictionary.
Inicialmente hacemos que la clase TWSIdentityDictionary utilice los mismos traits que utiliza TWSDictionary.
Luego para compartir comportamiento entre TWSDictionary y TWSIdentityDictionary creamos el trait TAbstractDictionary. Al momento de compartir comportamiento es necesario refactorizar muchos mensajes para que no utilicen variables de instancia directamente.
La Figura 24 muestra los cambios en el modelo.

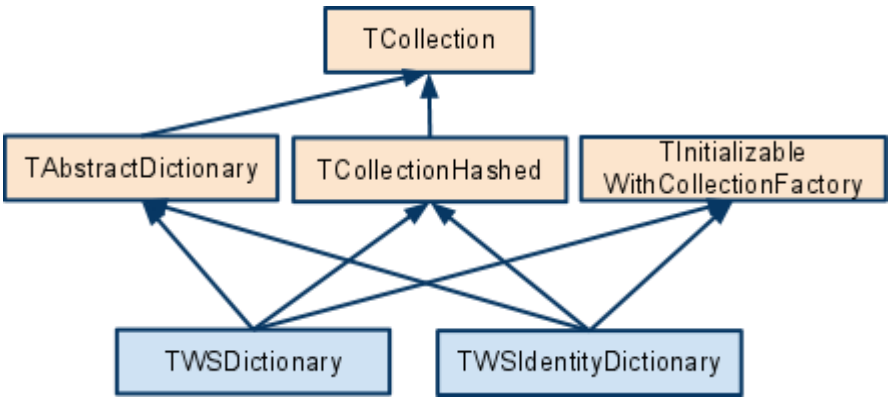

Figura 24 - Diagrama de traits y clases del modelo luego de la implementación de TWSIdentityDictionary

4.2.2.7.1 Métricas de Desarrollo

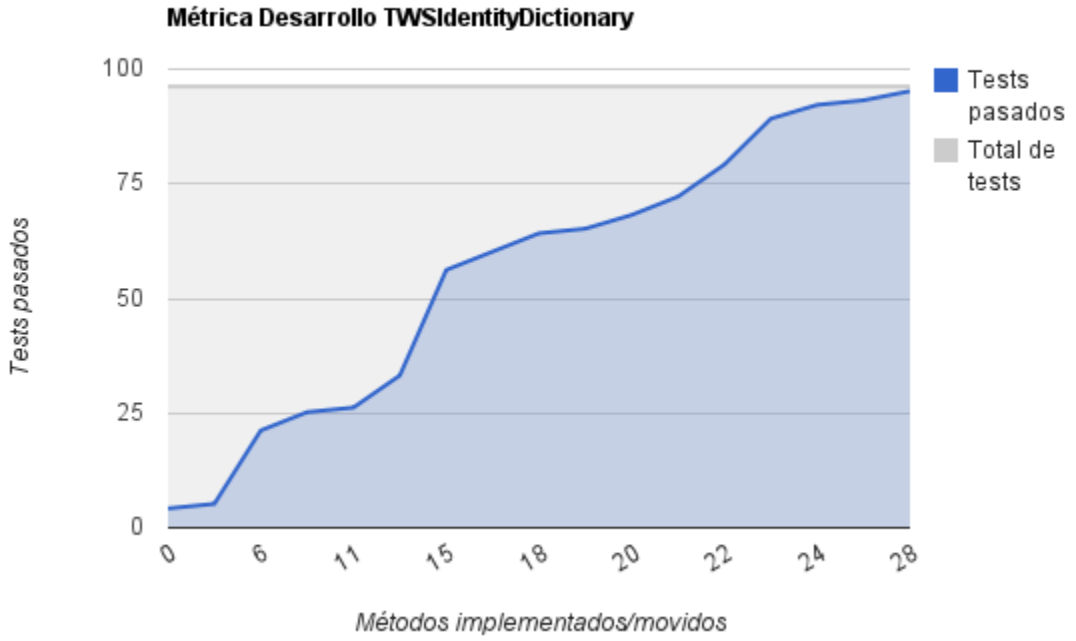

Figura 25 - Métrica de Desarrollo de la clase TWSIdentityDictionary

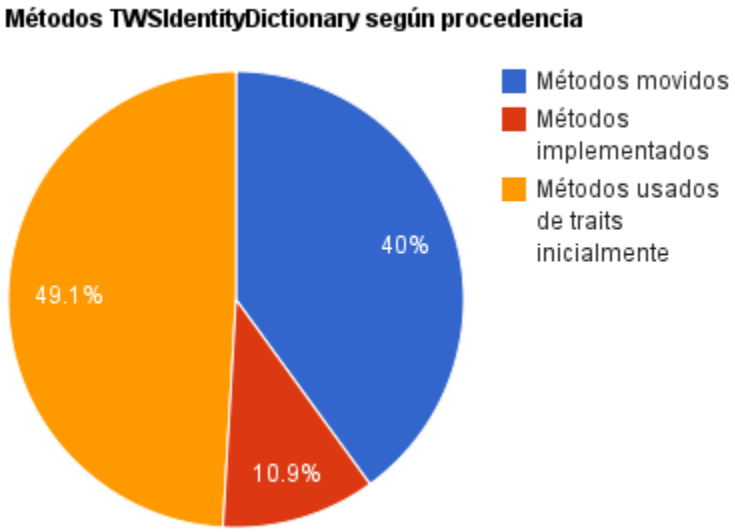

Figura 26 - Métrica de Métodos Según Procedencia de la clase TWSIdentityDictionary

Las figuras 25 y 26 muestran el esfuerzo de desarrollo de la clase TWSIdentityDictionary. Sólo fue necesario implementar el 10.9% de los métodos. El porcentaje de métodos implementados corresponde a un total de 6 métodos de los cuales 4 son métodos de acceso a variables de instancia. Esto indica un alto porcentaje de reutilización de código, y un bajo esfuerzo de implementación.

4.2.2.8 TWSBag

La implementación de Bag utiliza un Dictionary, por lo tanto ahora podemos implementar TWSBag utilizando un TWSDictionary.

No es necesario crear un nuevo trait para implementar TWSBag. La Figura 27 muestra el modelo resultante luego de la implementación.

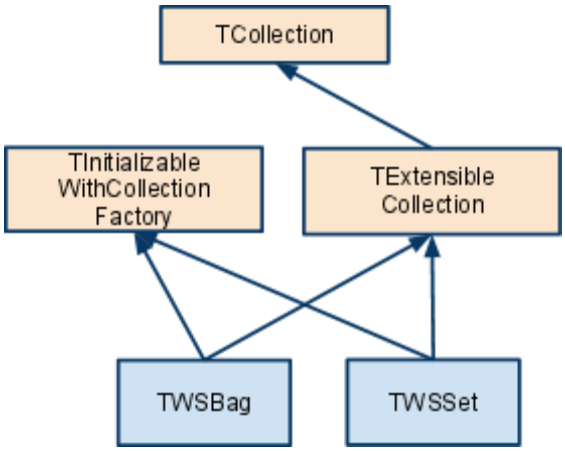

Figura 27 - Diagrama de traits y clases del modelo luego de la implementación de TWSBag

4.2.2.8.1 Métricas de Desarrollo

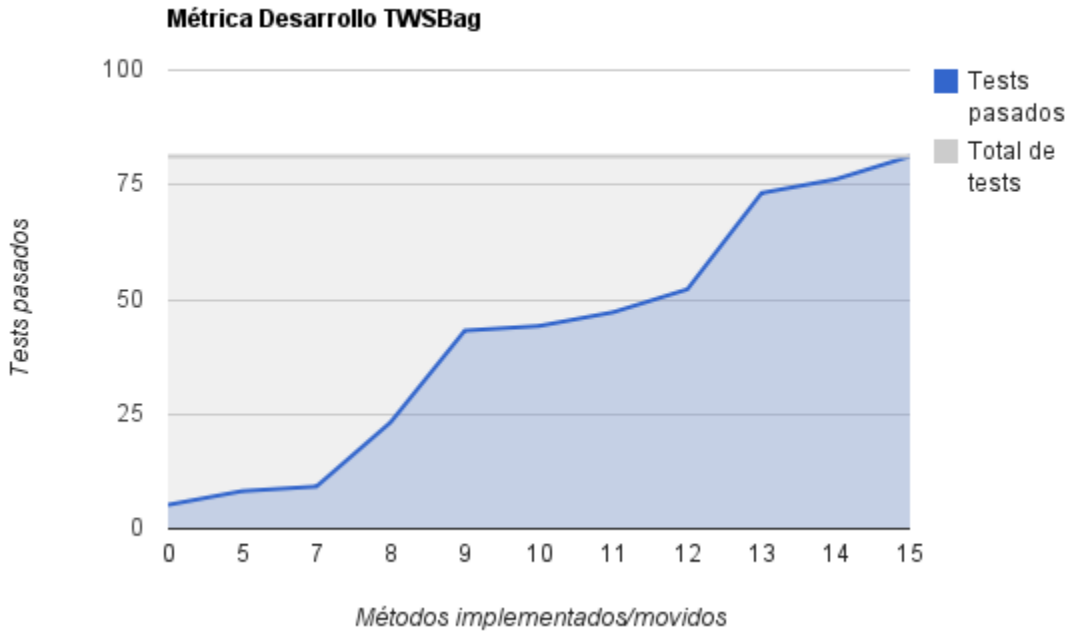

Figura 28 - Métrica de Desarrollo de la clase TWSBag

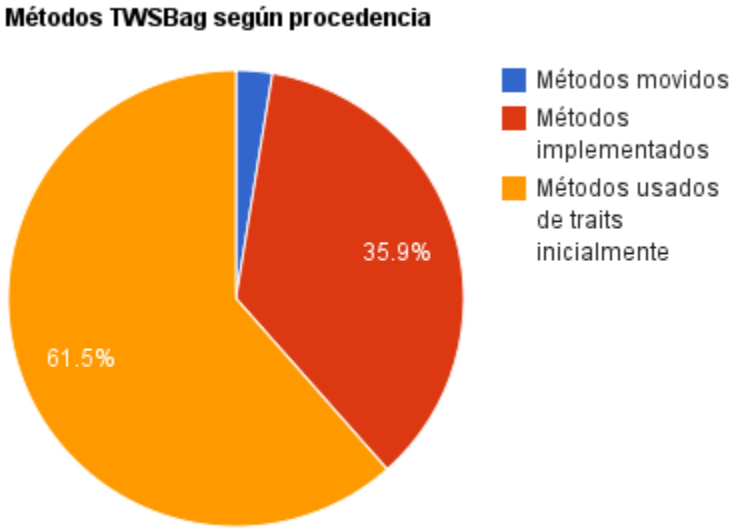

Figura 29 - Métrica de Métodos Según Procedencia de la clase TWSBag

Las figuras 28 y 29 muestran el esfuerzo de desarrollo de la clase TWSBag. Los métodos implementados corresponden a mejoras de performance aprovechando el hecho de que un Bag contiene elementos repetidos.

4.2.2.9 TWSInterval

El protocolo Interval solamente cumple con el protocolo SequencedReadableCollection del cual existen ya dos implementaciones en el modelo. Por otro lado, utiliza solo 3 enteros como variables de instancia para su implementación en Pharo. Por lo tanto puede ser el siguiente protocolo por implementar.

Comenzamos utilizando el trait TSequencedReadableCollection, pero luego descubrimos que podíamos utilizar la implementación TSecuencedReadableArrayedCollection. Decidimos entonces cambiar el nombre de este trait, y el de TSequencedCollection por TSequencedReadableFixedSizeCollection y TSequencedFixedSizeCollection respectivamente. La Figura 30 muestra la parte del modelo relevante afectada por la implementación de esta clase.

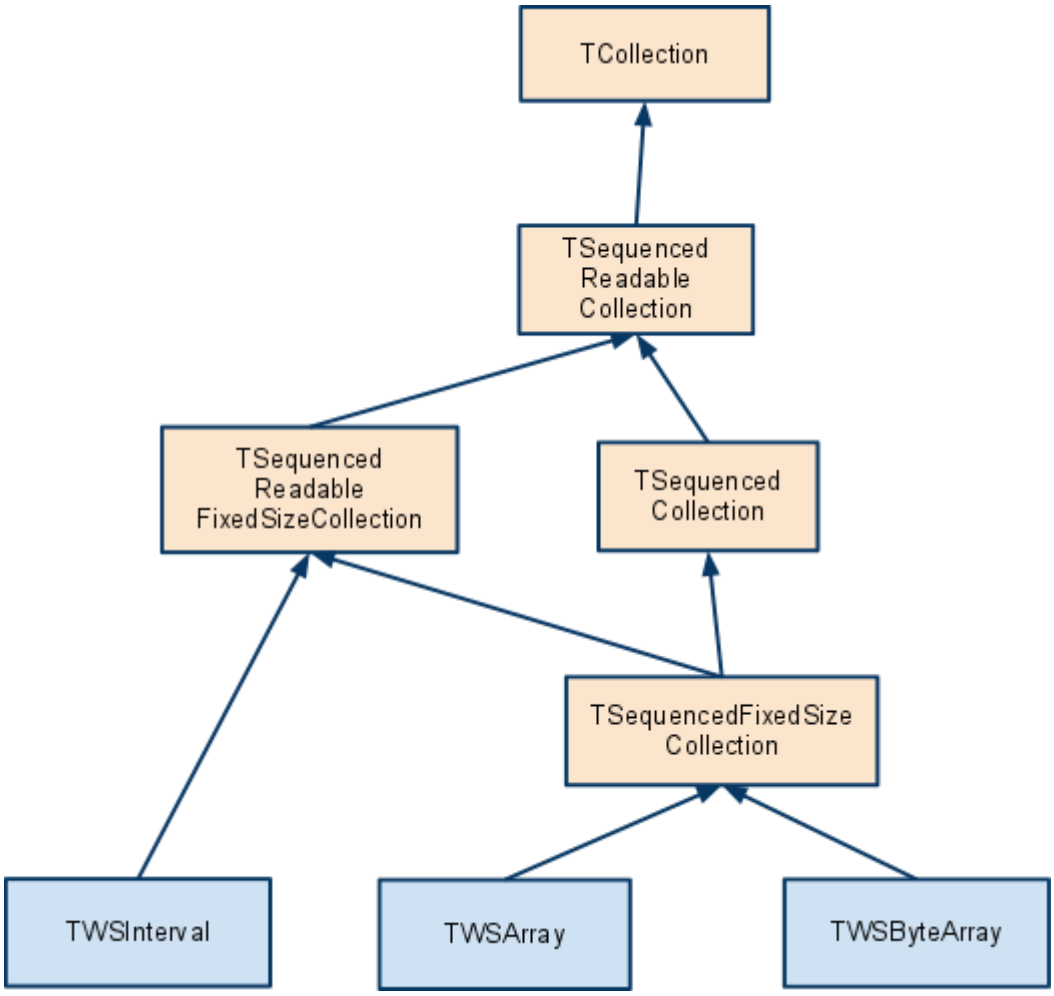

Figura 30 - Diagrama de traits y clases del modelo luego de la implementación de TWSInterval

4.2.2.9.1 Métricas de Desarrollo

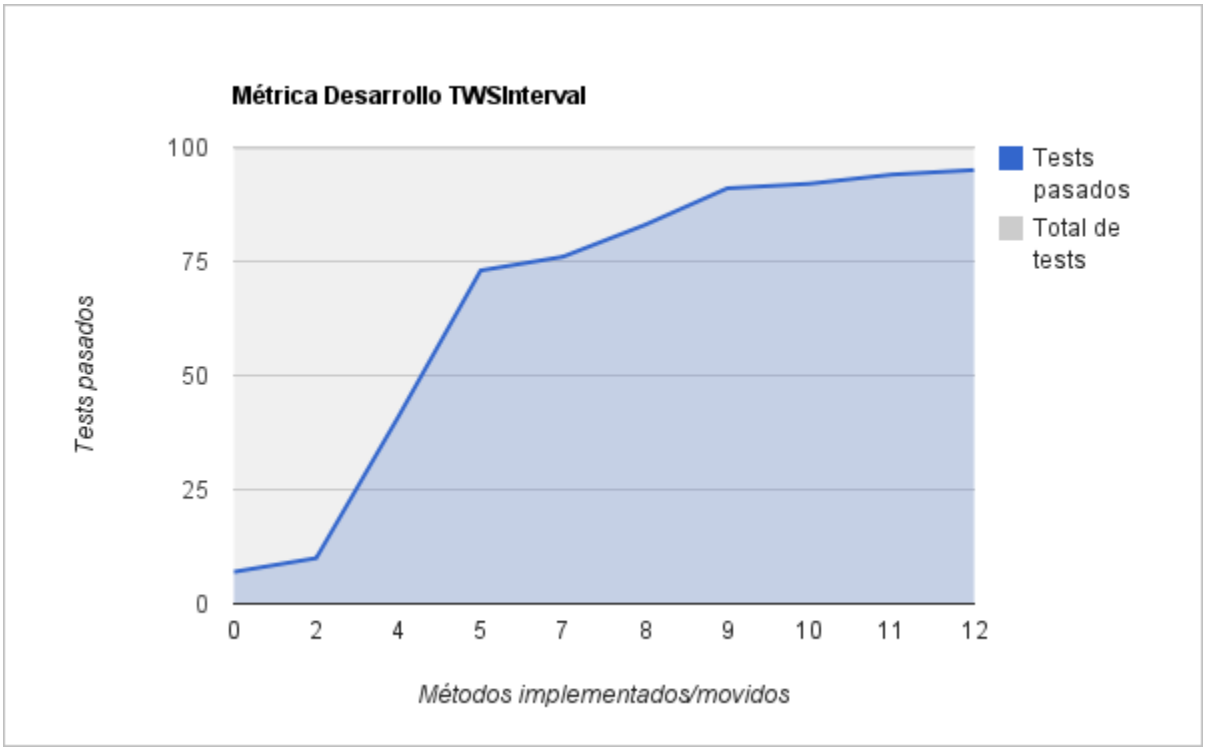

Figura 31 - Métricas de Desarrollo de la clase TWSInterval

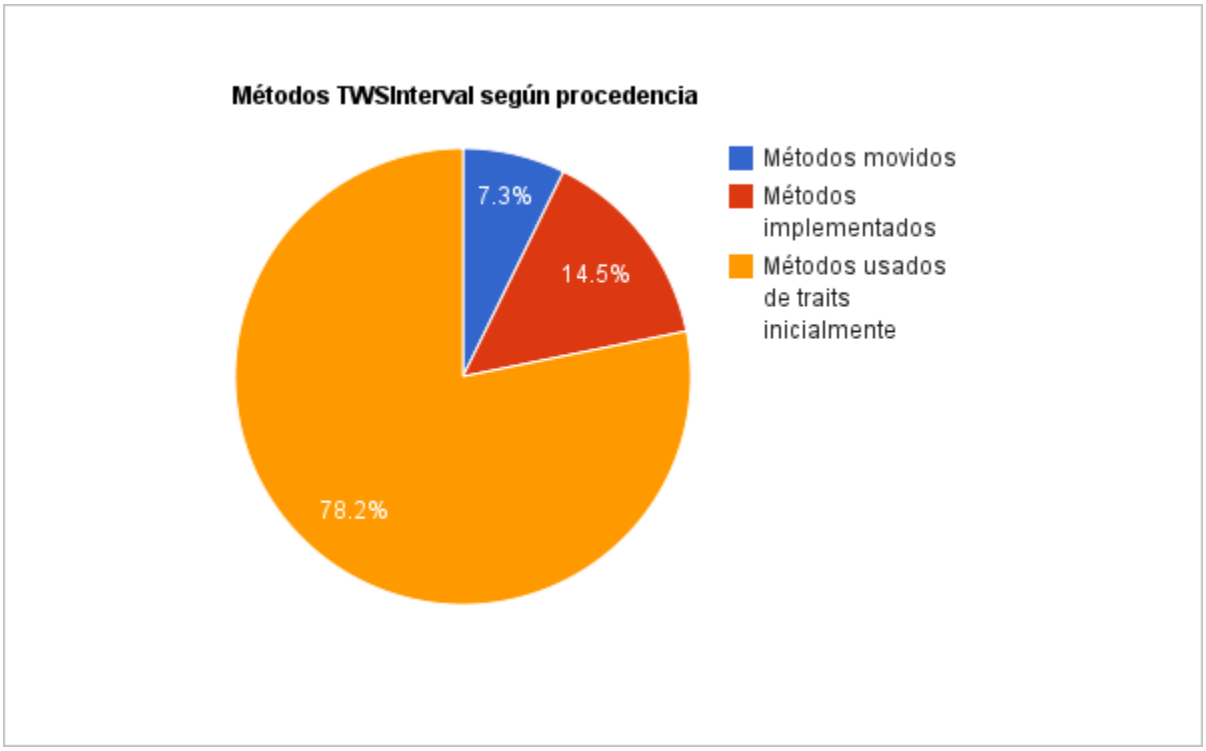

Figura 32 - Métricas de Métodos Según Procedencia de la clase TWSInterval

Las figuras 31 y 32 muestran el esfuerzo de desarrollo de la clase TWSInterval. Observamos un gran porcentaje de métodos compartidos inicialmente, y un bajo porcentaje de métodos implementados. Podemos decir entonces que el esfuerzo de desarrollo de TWSInterval es bajo.

4.2.2.10 TWSString

Las clases de los protocolos son las que restan implementar en este proceso. Decidimos continuar implementado el protocolo String.
La implementación de String realizada es utilizando una *variable byte subclass* de Object.
Según el ANSI el protocolo String cumple con el protocolo SequencedCollection y ReadableString. Actualmente tenemos dos implementaciones del protocolo SequencedCollection: la implementación FixedSize y la Growable. También contamos con la posibilidad de utilizar la Implementación de SequencedCollection que es común a ambas, pero de ese modo no se utilizan muchos métodos.
Encontramos que la implementación que debemos usar para TWSString es la FixedSize. Por lo tanto hacemos que la clase utilice TSequencedFixedSizeCollection. Utilizando esta implementación contamos con 61 métodos inicialmente. Y como se aprecia en las métricas el desarrollo se acelera notoriamente.
Los métodos pertenecientes al protocolo ReadableString se implementan directamente en TWSString, por lo tanto no creamos ningún trait nuevo.
La Figura 33 muestra los cambios realizados al modelo.

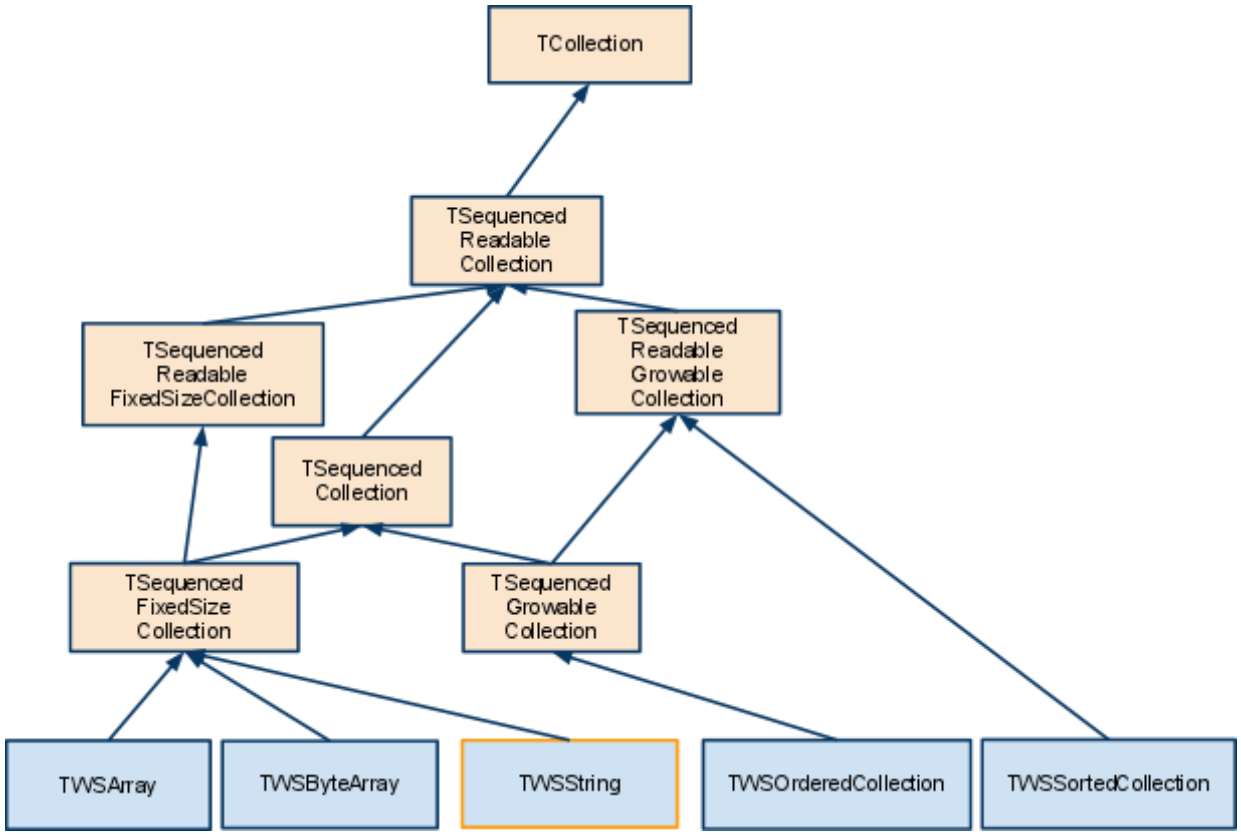

Figura 33 - Diagrama de traits y clases que muestra un recorte del modelo luego de la implementación de TWSString

4.2.2.10.1 Métricas de Desarrollo

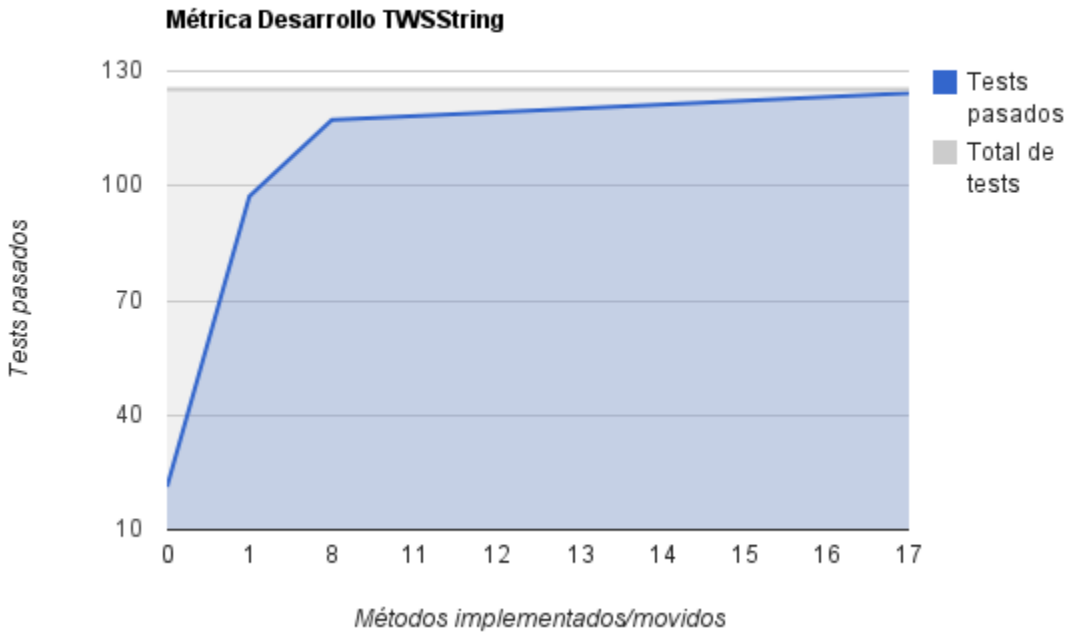

Figura 34 - Métrica de Desarrollo de la clase TWSString

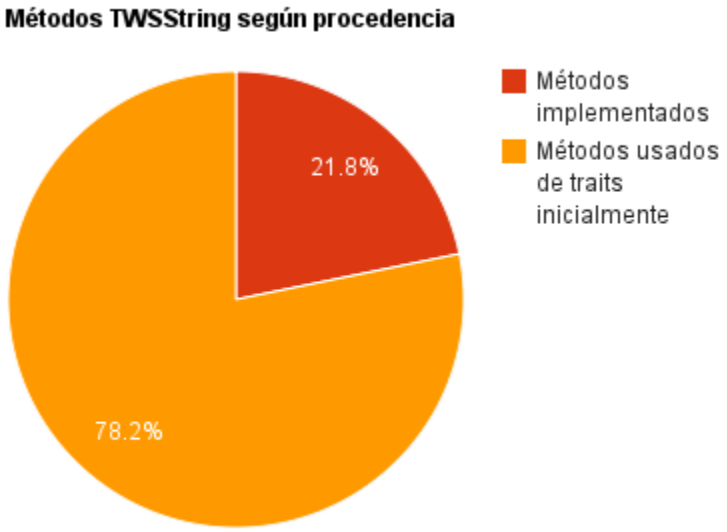

Figura 35 - Métrica de Métodos Según Procedencia de la clase TWSString

Las figuras 34 y 35 muestran el esfuerzo de desarrollo de la clase TWSString. A medida que avanzamos en el desarrollo del modelo observamos grandes porcentajes de métodos compartidos a través de traits inicialmente, y esto indica que los métodos ya compartidos anteriormente son reutilizados. Podemos decir que el esfuerzo de desarrollo de la clase es bajo, ya que sólo fue necesario implementar el 21.8% de los métodos.

4.2.2.11 TWSSymbol

El último protocolo que nos resta implementar es el de Symbol.
La implementación de Symbol realizada utiliza una *variable byte subclass* de Object.
El primer problema con Symbol es la creación de instancias. El ANSI no especifica un protocolo para la creación de instancias de Symbol. La particularidad de la creación de instancias de Symbol es que no siempre se crea una nueva instancia cuando es solicitada. Un objeto de la clase Symbol es único, lo que significa que no puede existir otro igual o sea con la misma secuencia de caracteres. Lo que se hace cuando se quiere un símbolo que ya fue creado, es devolver la instancia ya creada.
El protocolo utilizado para la creación de instancias de Symbol es el siguiente:

```
TWSSymbol>>withAll:
```

Este mensaje recibe una colección que cumple con el protocolo ReadableString del ANSI y devuelve un objeto instancia de TWSSymbol. El objeto devuelto contiene exactamente los

elementos de la colección pasada como parámetro en igual orden. El objeto devuelto es único, esto quiere decir que no existe otro con la misma secuencia de caracteres. O dicho de otra manera, dado el objeto devuelto s1, no existe otro s2 tal que: `(s1 == s2) not` devuelva `true`.

Dejando de lado la creación de instancias, el protocolo Symbol cumple con el protocolo ReadableString, el cual a su vez cumple con SequencedReadableCollection. Dado que inicialmente no existe implementación del protocolo ReadableString en un trait, ya que la única implementación se encuentra en la clase TWSString, utilizamos la implementación del protocolo SequencedReadableCollection correspondiente al trait TSequencedReadableFixedSizeCollection en TWSSymbol en primera instancia.

Método `#asLowercase`

Detallaremos el desarrollo de este método ya que se presentaron alternativas de implementación no convencionales.

Al momento de implementar el método `#asLowercase` en TWSSymbol nos encontramos con diferentes alternativas.
En Pharo, el método `#asLowercase` se encuentra implementado en la clase String, y la clase Symbol lo hereda ya que es subclase de String.

La primera alternativa encontrada es compartirlo a través de un trait entre las clases TWSSymbol y TWSString. El problema de compartir el método `#asLowercase` a través de un trait es que la implementación accede a una variable de clase directamente. Análogamente a cuando se necesita compartir un método a través de un trait que accede a variables de instancia directamente, refactorizamos el método para que acceda a la variable a través de un método de acceso, y modelamos esto como un requerimiento en el trait, salvo que en este caso el requerimiento es a nivel de clase.
Pero entonces debemos analizar si es necesario tener una variable de clase en tanto en TWSString como en TWSSymbol.
Veamos la implementación de `#asLowercase` en TWSString.

`TWSString>>asLowercase`

```
^ self copy asString translateWith: LowercasingTable
```

Luego de observar la implementación surge otra alternativa que no involucra la creación de un trait, y tampoco necesita la creación de una nueva variable de instancia. La solución es implementar `#asLowercase` en TWSSymbol de la siguiente manera:

`TWSSymbol>>asLowercase`

```
^ self copy asString asLowercase
```

Esta misma implementación podría utilizarse en la clase Symbol de Pharo, pero la herencia permite reutilizar la implementación de String.
Esta implementación es posible porque el método `#asLowercase` en TWSSymbol devuelve un TWSString.

Métodos de Comparación

Detallaremos el desarrollo de este método ya que se presentaron alternativas de implementación no convencionales.

Los mensajes de comparación pertenecientes al protocolo ReadableString pueden compartir implementación entre TWSString y TWSSymbol.
En la implementación de Pharo, estos métodos son compartidos por las clases String y Symbol a través de herencia. A continuación mostramos la implementación de Pharo del mensaje `#<`

```
String>>< aString

	^ (self compare: self with: aString collated: AsciiOrder) = 1
```

Como vemos la implementación utiliza una variable de clase llamada AsciiOrder. No podemos compartir un método que accede a una variable de clase directamente a través de un trait. Para solucionar esto tenemos principalmente tres alternativas:

1. Acceder al valor de AsciiOrder a través de una variable global
2. Crear un requerimiento en el trait, que se pueda utilizar como método para acceder al objeto AsciiOrder.
3. Encapsular en un mensaje la implementación para acceder al objeto AsciiOrder. Y teniendo en el trait una implementación por defecto.

```
TReadableStringFixedSize>>< aString

^(self compare: self with: aString collated: self asciiOrder) = 1

"Donde asciiOrder es un requerimiento"
```

Analizando las ventajas y desventajas de las alternativas, observamos que:

- La primera alternativa no obliga a las clases que utilizan el trait a implementar un requerimiento.
- Dado que el orden ascii no parece ser algo que vaya a cambiar, la primera opción evita repetir código, ya que si el orden ascii siempre es el mismo, todas las clases que utilicen el trait de la segunda opción implementarán el requerimiento de igual manera.

- La segunda opción parece ser más flexible ya que permite implementar el requerimiento accediendo a una variable de clase, a una variable global, o creando el objeto en el momento.
- Por último, la tercera alternativa tiene las ventajas de las dos anteriores: No obliga a implementar un requerimiento, y al mismo tiempo ofrece la posibilidad de sobrescribir la implementación por defecto, brindando la flexibilidad de la segunda opción.

Basándonos en este análisis elegimos la tercera opción, utilizando como implementación por defecto acceder al objeto asciiOrder a través de un método de clase de TWSString.

4.2.2.11.1 Métricas de Desarrollo

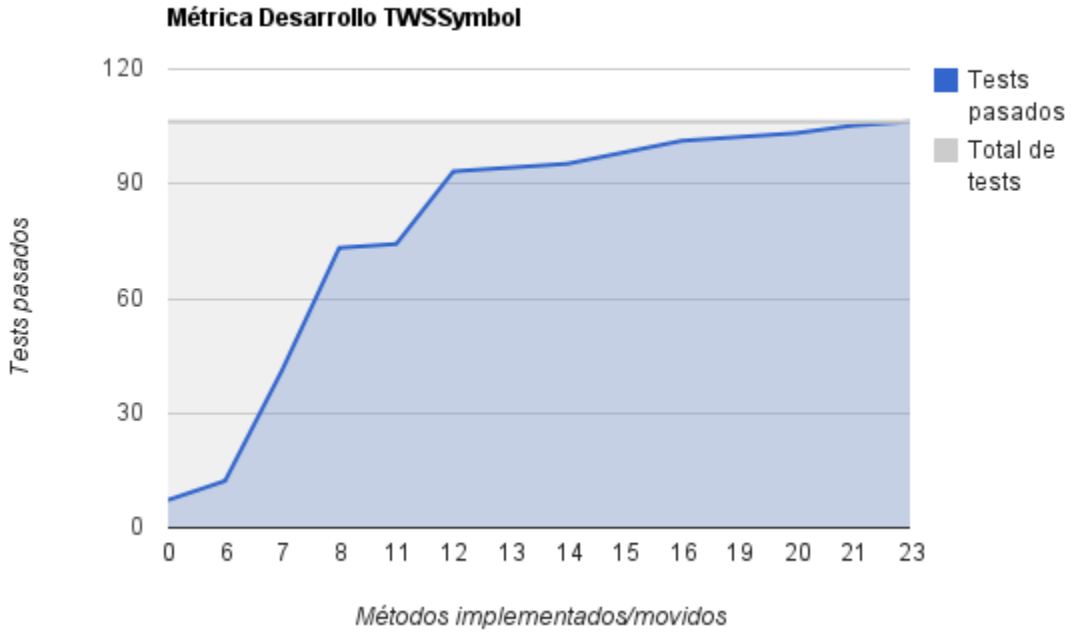

Figura 36 - Métrica de Desarrollo de la clase TWSSymbol

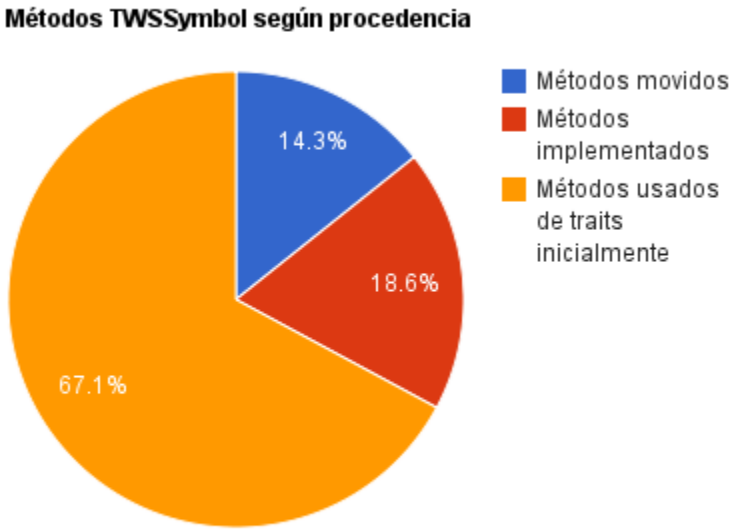

Figura 37 - Métrica de Métodos Según Procedencia de la clase TWSSymbol

Las figuras 36 y 37 muestran el esfuerzo de desarrollo de la clase TWSSymbol. Al igual que en TWSString, podemos observar un esfuerzo de desarrollo bajo, ya que sólo se implementaron el 18.6% de los métodos.

4.2.2.12 Métricas Totales del Desarrollo

A continuación compararemos las métricas de desarrollo de cada clase en el orden en que fueron implementadas. Para medir las características del desarrollo de una clase tenemos en cuenta tres factores:

- **Los métodos implementados**: Cuando implementamos un método directamente en la clase que estamos desarrollando
- **Los métodos movidos**: Cuando movemos un método desde un trait o una clase hacia un trait, de manera que pueda ser compartido con la clase que estamos desarrollando.
- **Los métodos utilizados inicialmente desde traits**: Son los métodos que utilizamos al declarar que una clase utiliza un trait.

Para cada uno de estos factores medimos la cantidad de métodos y el porcentaje con respecto al total de métodos de una clase.

4.2.2.12.1 Métodos Implementados

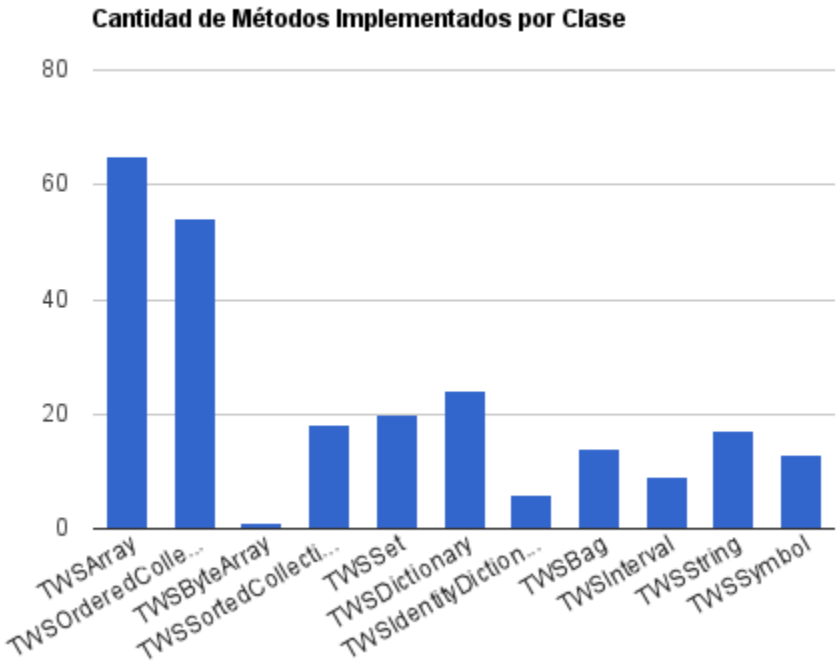

Figura 38 - Cantidad de Métodos Implementados por Clase

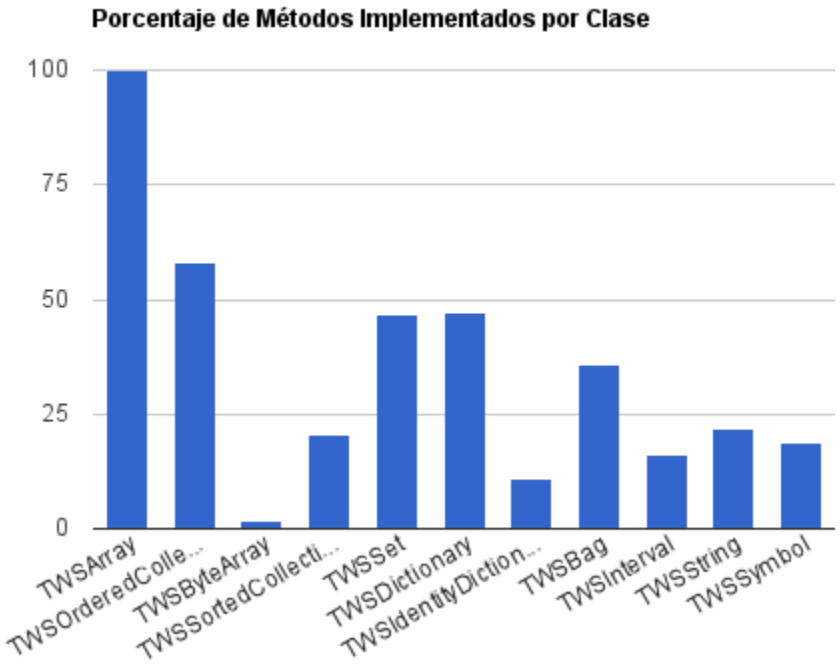

Figura 39 - Porcentaje de Métodos Implementados por Clase

Como muestran las figuras 38 y 39, en general, podemos notar que la cantidad de métodos implementados va decreciendo a medida que vamos implementando más clases. Y es una buena señal de que cada vez estamos implementando menos y compartiendo más métodos.

4.2.2.12.2 Métodos Movidos

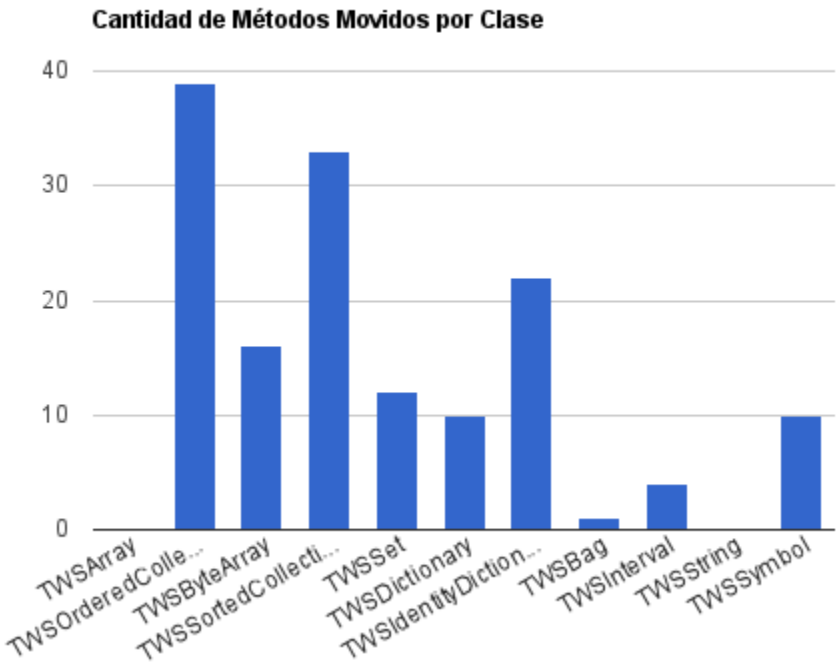

Figura 40 - Cantidad de Métodos Movidos por Clase

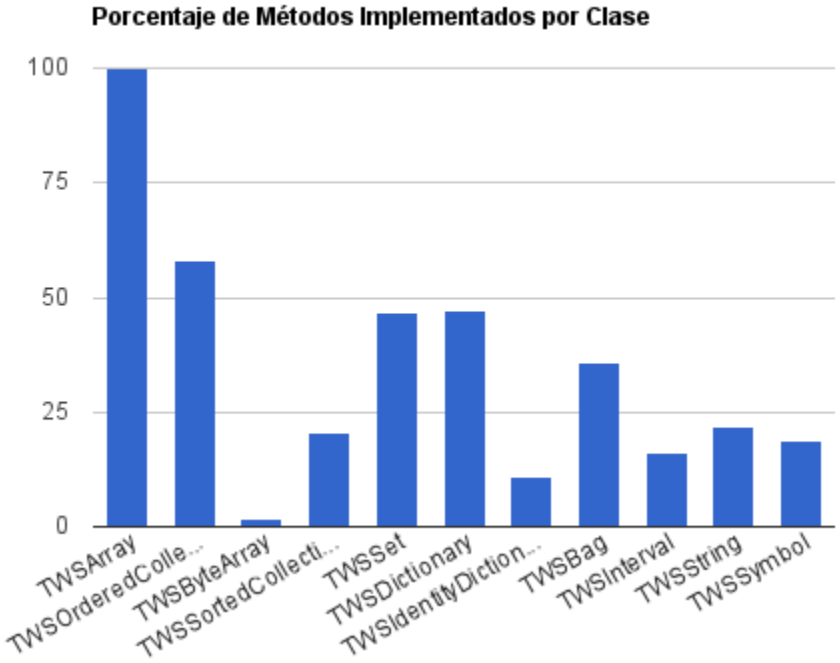

Figura 41 - Porcentaje de Métodos Movidos por Clase

Las figuras 40 y 41 muestran que la cantidad de métodos movidos decrece a medida que el modelo madura. Esto también puede verse como una señal de que el modelo evoluciona favorablemente, siendo menor el esfuerzo de desarrollo a medida que avanzamos. Esto se debe al hecho de que a medida que el modelo madura, una clase nueva inicialmente utiliza muchos métodos, lo cual hace que no sea necesario mover o implementar tantos métodos.

4.2.2.12.3 Métodos Utilizados Inicialmente Desde Traits

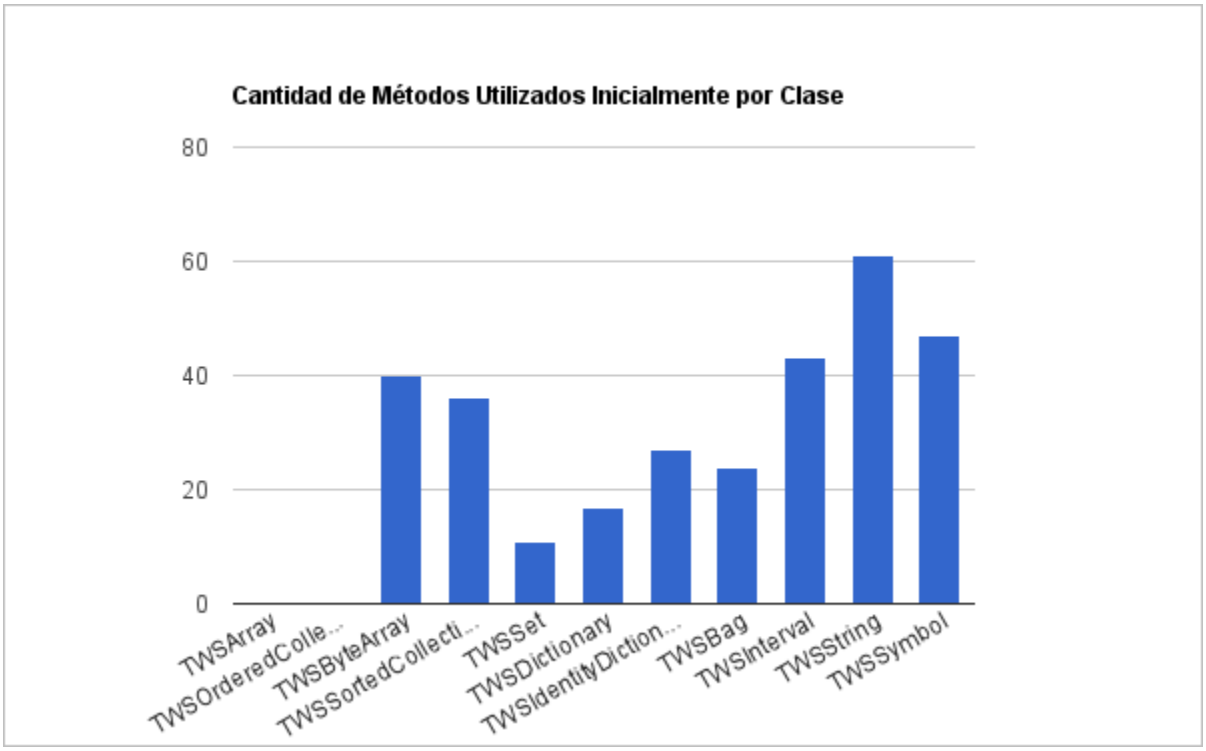

Figura 42 - Cantidad de Métodos Utilizados Inicialmente por Clase

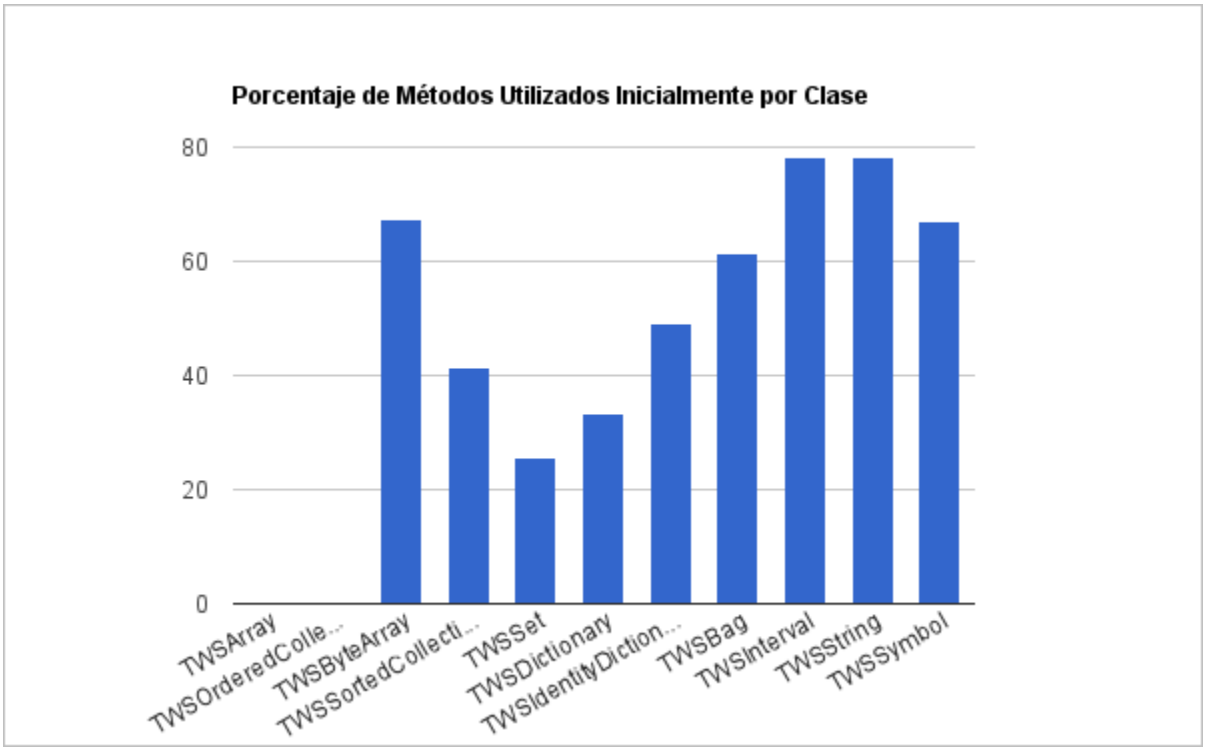

Figura 43 - Porcentaje de Métodos Utilizados Inicialmente por Clase

Las figuras 42 y 43 muestran los métodos utilizados inicialmente clase por clase. Si tenemos en cuenta que la clase TWSByteArray es un caso especial, observamos que en general la cantidad y el porcentaje de métodos que inicialmente se utiliza se va incrementando, y esto es una señal de que el modelo va evolucionando de manera correcta.
TWSByteArray es un caso particular porque comparte casi toda su implementación con TWSArray. De hecho su protocolo es idéntico.
Otro caso para analizar es el de TWSSet, ya que tiene el menor porcentaje de métodos compartidos inicialmente. TWSSet comparte su protocolo con TWSSortedCollection, pero sus implementaciones son diferentes.

4.2.2.13 Refactorización Final

En un refactoring previo, explicado dentro de la sección TWSSortedCollection, creamos el trait TSequencedGrowableCollection, que finalmente sería utilizado por una única clase. Esto hace que pierda un poco el sentido, ya que no comparte comportamiento con ninguna otra clase. Es por esto que el trait es eliminado y la clase que depende de éste, ahora utilizará los traits de los cuales se componía.

Finalizado el desarrollo encontramos 3 traits que eran utilizados únicamente por OrderedCollection y SortedCollection. Decidimos entonces reemplazar a estos 3 traits, por uno solo que contenga el comportamiento de los 3. Esto se hace para obtener un modelo de menor complejidad.

4.2.3 Modelo Final Resultante

En la Figura 44 observamos el nuevo modelo de Collections que utiliza únicamente traits como mecanismo para compartir comportamiento. Este modelo es resultado del desarrollo anteriormente explicado.

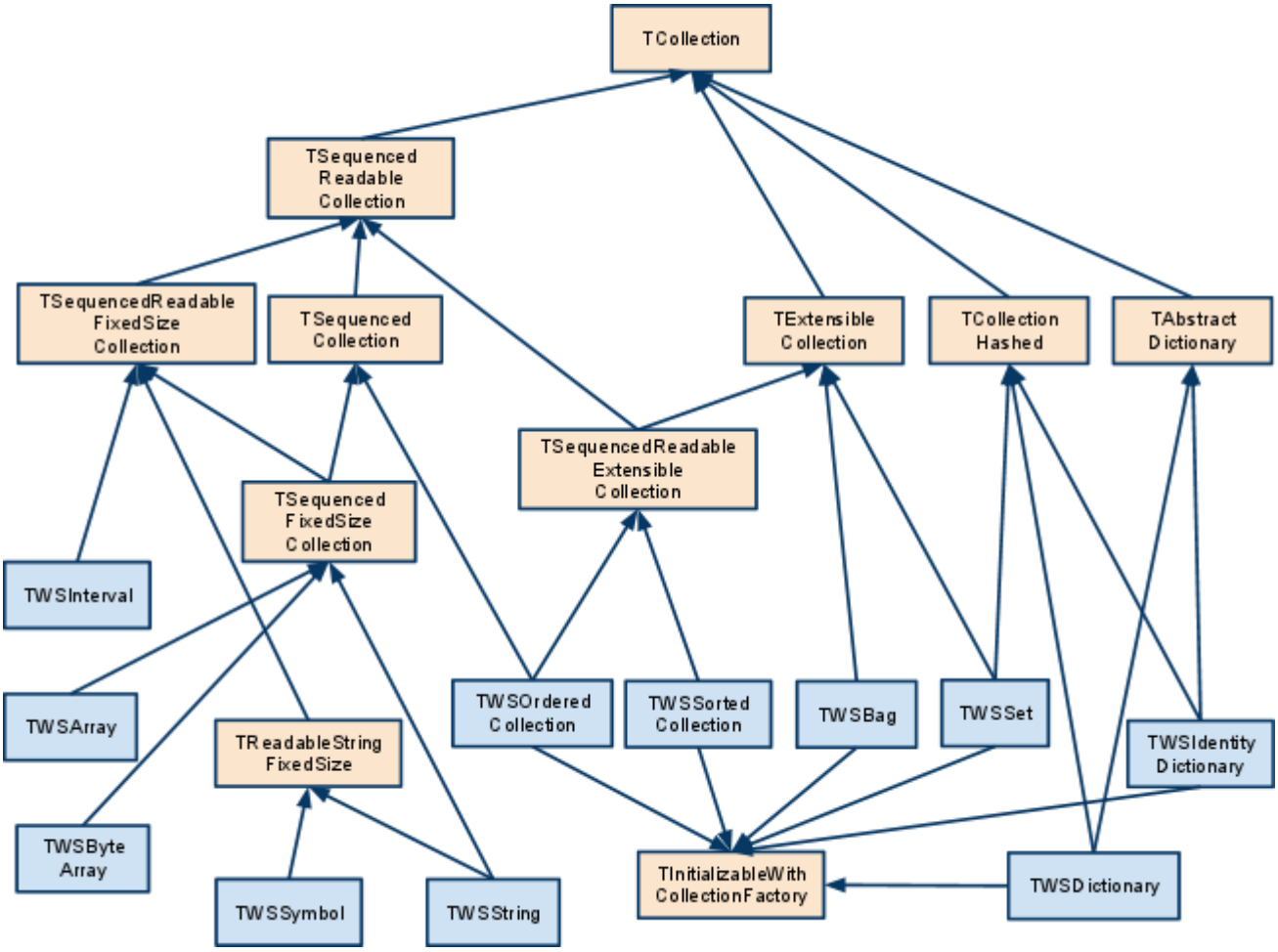

Figura 44 - Modelo Final Utilizando Traits sin Subclasificación de los Protocolos de Collection

4.2.3.1 Métricas Del Modelo Final

A continuación mostraremos métricas del modelo resultante con el fin de compararlo con el modelo actual de Pharo, y además, evaluar sus ventajas y desventajas.

Cantidad de traits: 11

Cantidad de clases: 11
Cantidad total de métodos de traits: 138
Cantidad total de métodos implementados en clases: 118
Cantidad de métodos cancelados: 0
Cantidad de composiciones de traits con métodos excluidos: 0
Cantidad total de líneas de código implementadas en clases: 612
Cantidad total de líneas de código implementadas en traits: 702
Longitud de cadena más larga de usos: 4

4.2.3.1.1 Métricas por clase

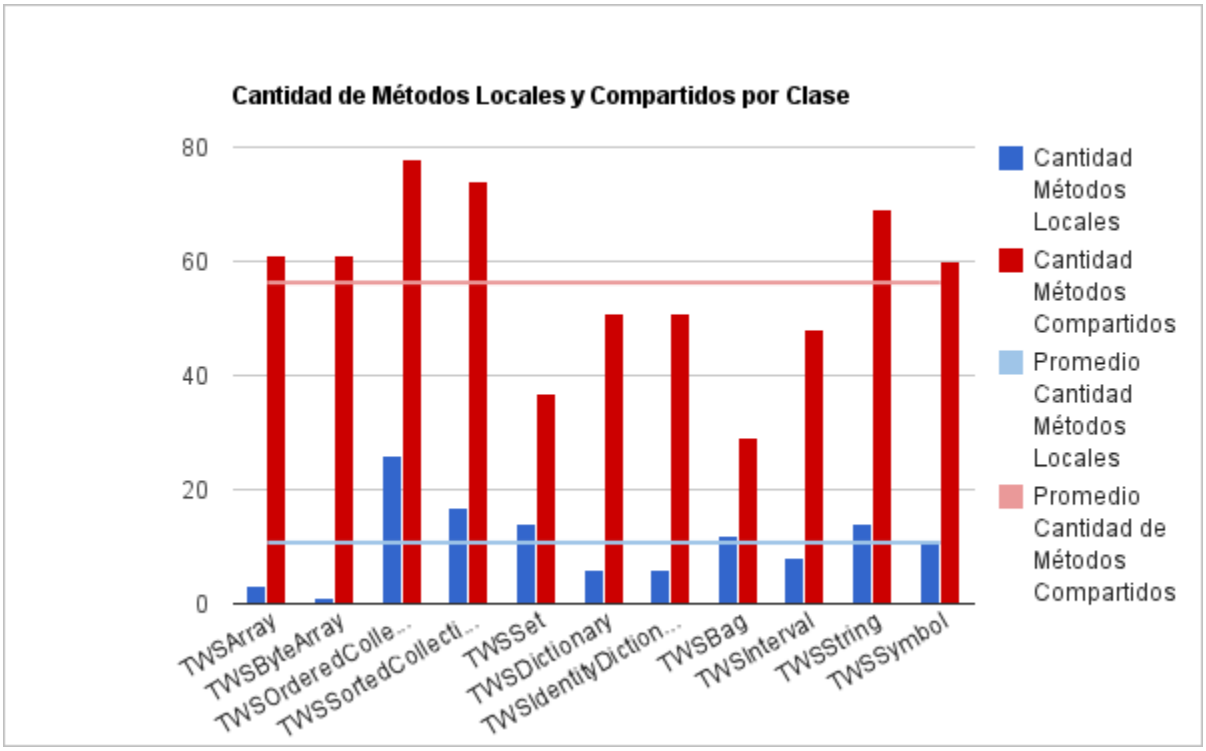

Figura 45 - Cantidad de Métodos Locales y Compartidos por Clase

En la Figura 45, debemos considerar que la cantidad total de métodos del modelo no es la sumatoria de los métodos de cada clase. Esto se debe a que existen clases que comparten métodos, y al sumar la cantidad de métodos de una clase con otra, se estaría contando dos veces cada método compartido.

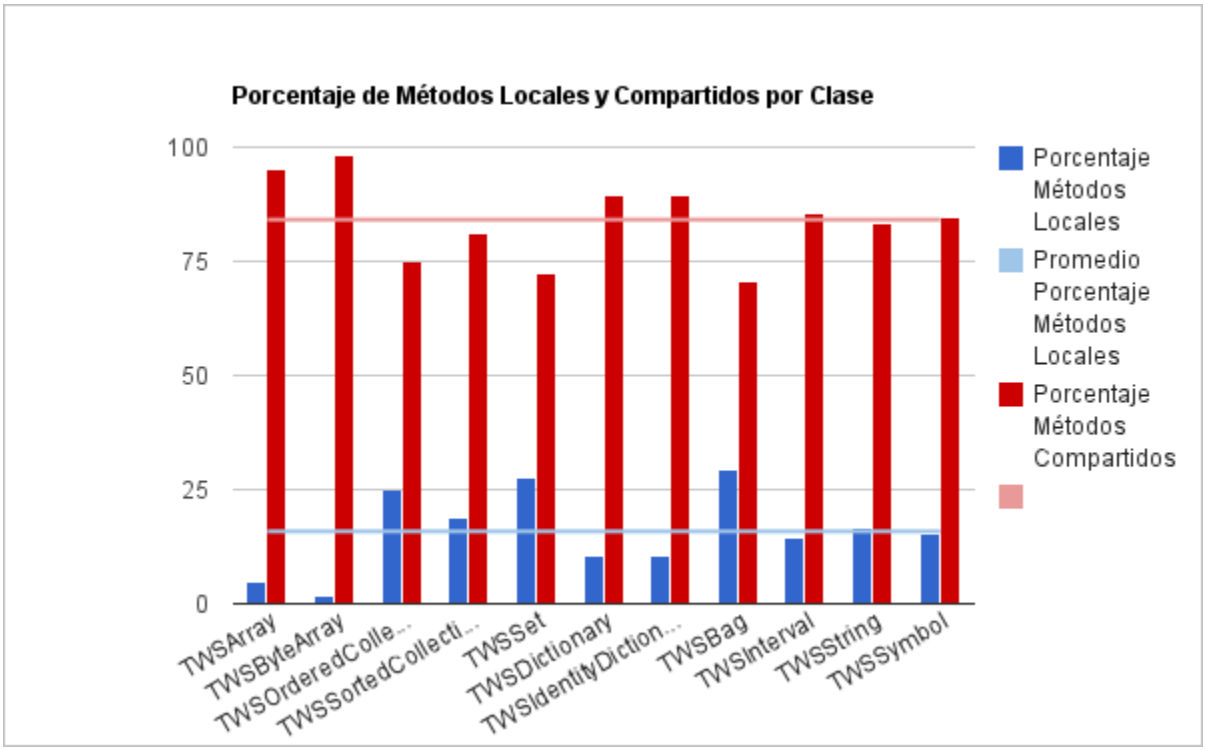

Figura 46 - Porcentaje de Métodos Locales y Compartidos por Clase

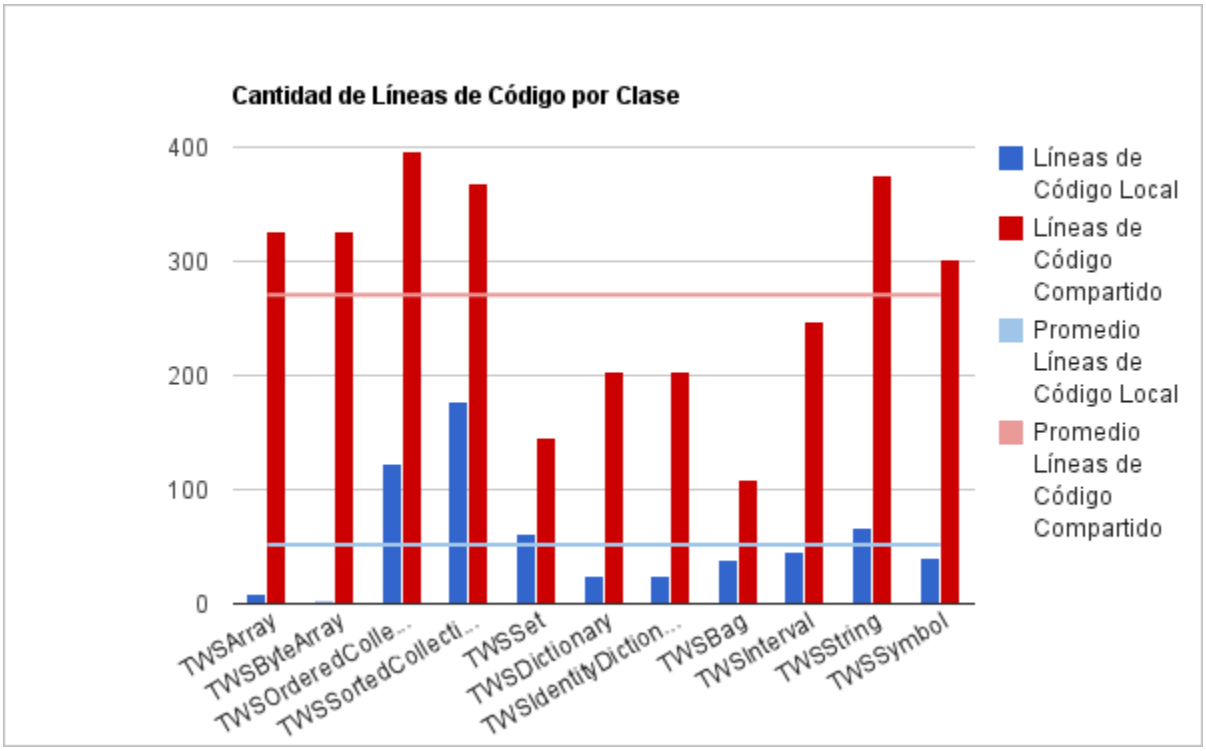

Figura 47 - Cantidad de Líneas de Código por Clase

Para la Figura 47, vale la misma aclaración que para la cantidad de métodos, no es posible obtener la cantidad total de líneas de código sumando las de cada clase ya que se comparten líneas de código entre clases.

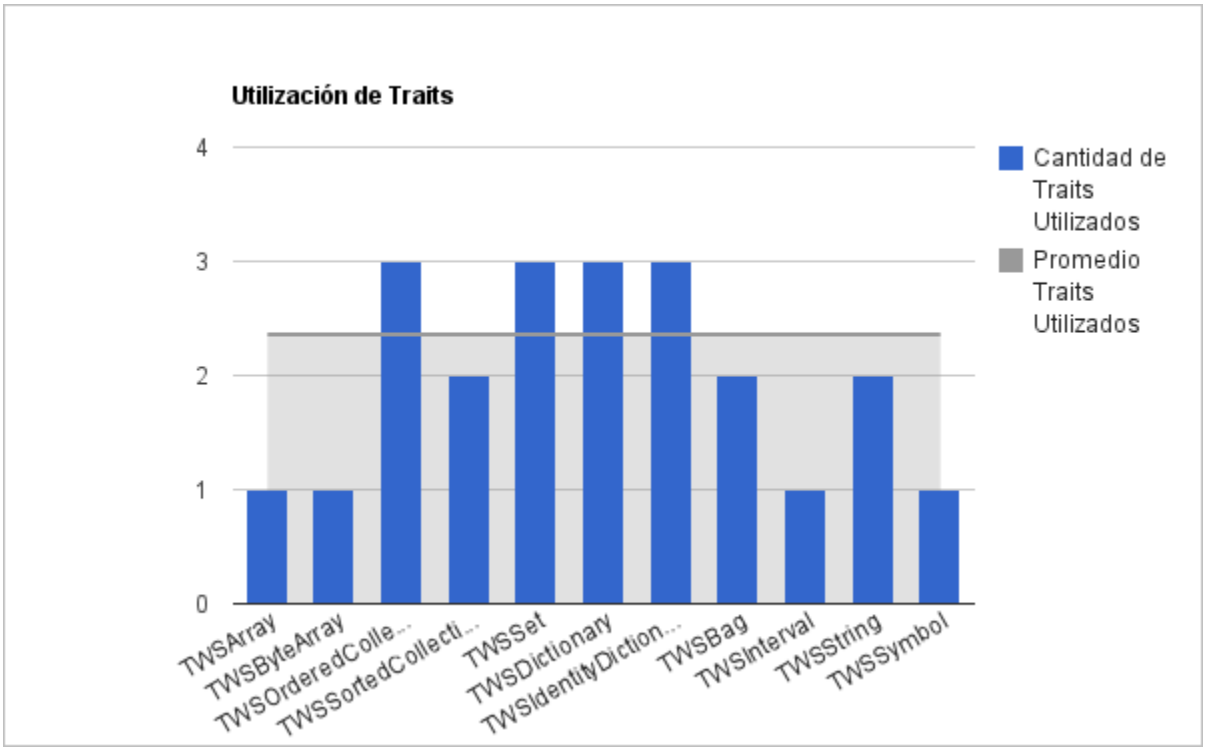

Figura 48 - Utilización de Traits por Clase

La Figura 48 muestra los traits utilizados en forma directa. No tiene en cuenta los utilizados indirectamente.

4.2.3.1.2 Métricas sobre Traits

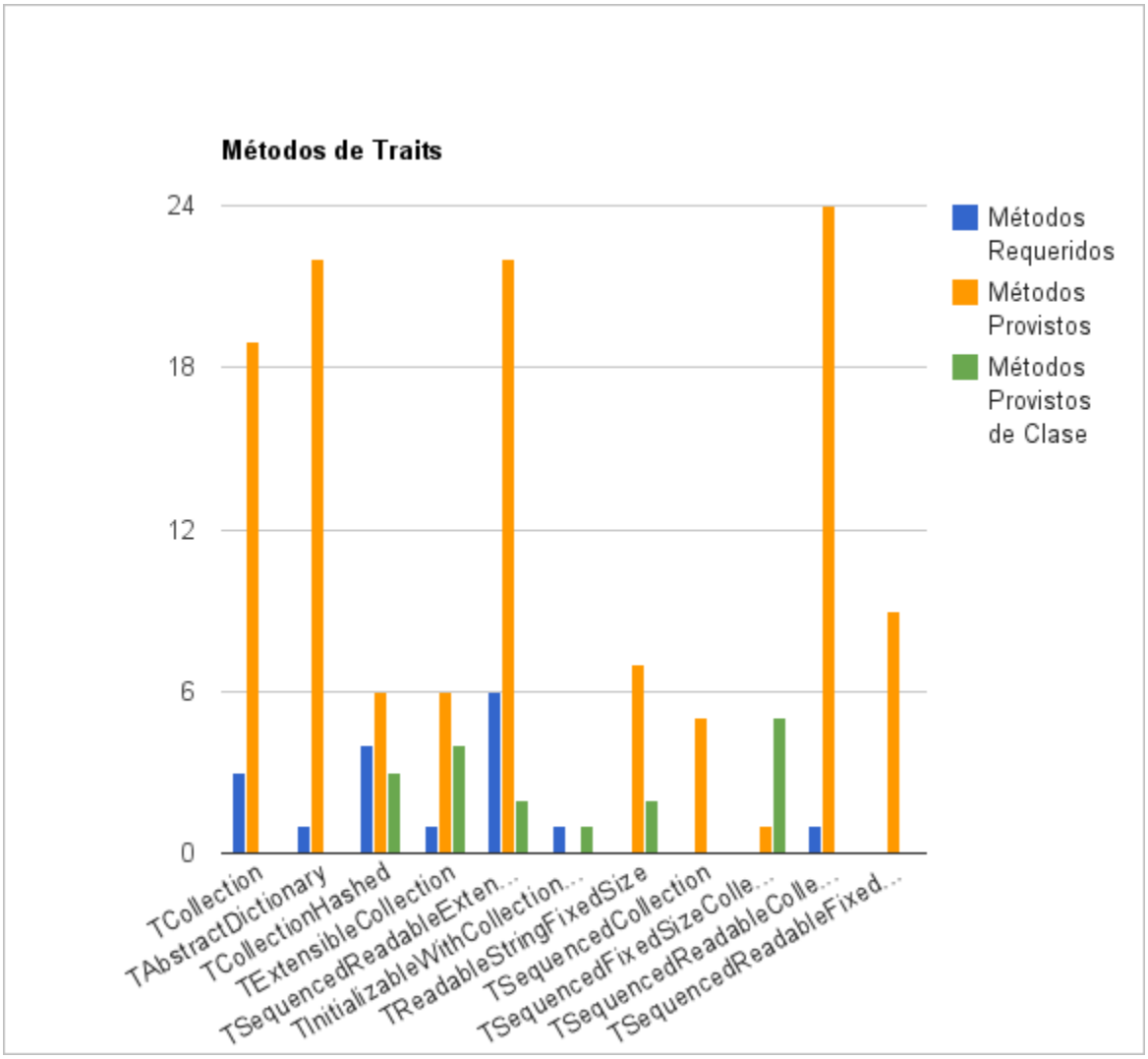

Figura 49 - Tipos de Métodos de Traits

Dado que no existen métodos de clase requeridos, no se especifican en la Figura 49. Los métodos provistos y requeridos mostrados en el gráfico, son aquellos que el trait implementa, y no aquellos que el trait utiliza a través de otro trait.

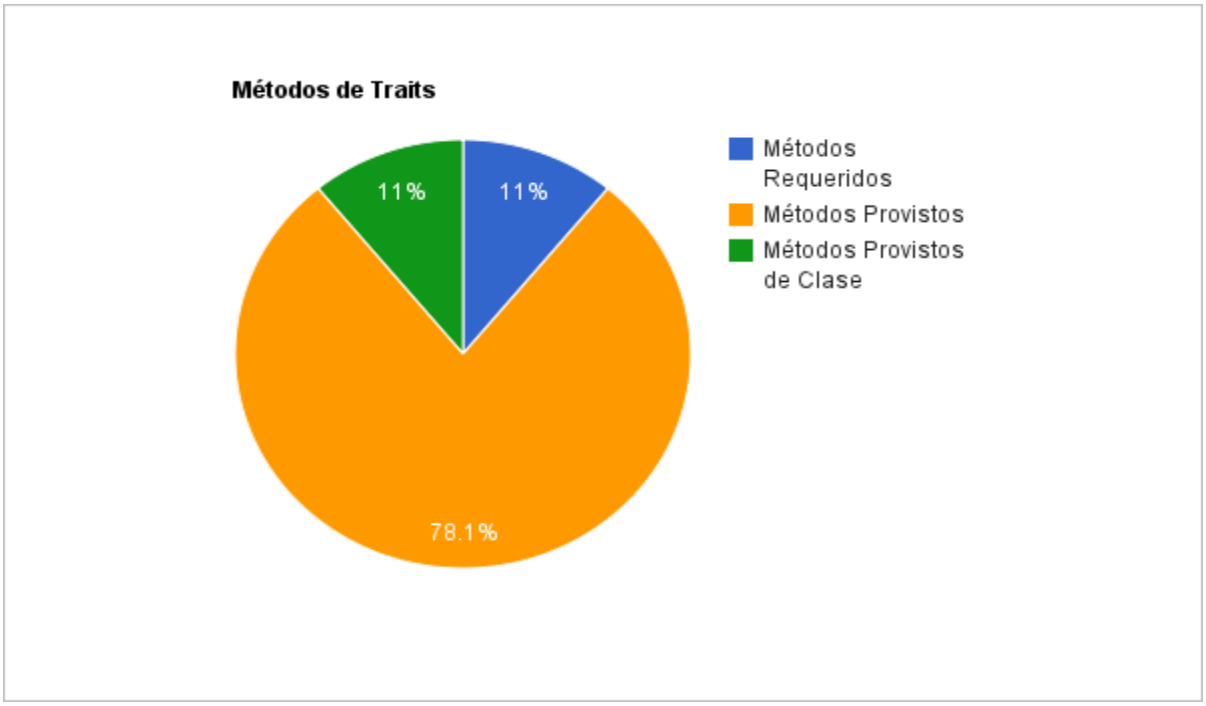

Figura 50 - Porcentajes de Tipos de Métodos de Traits

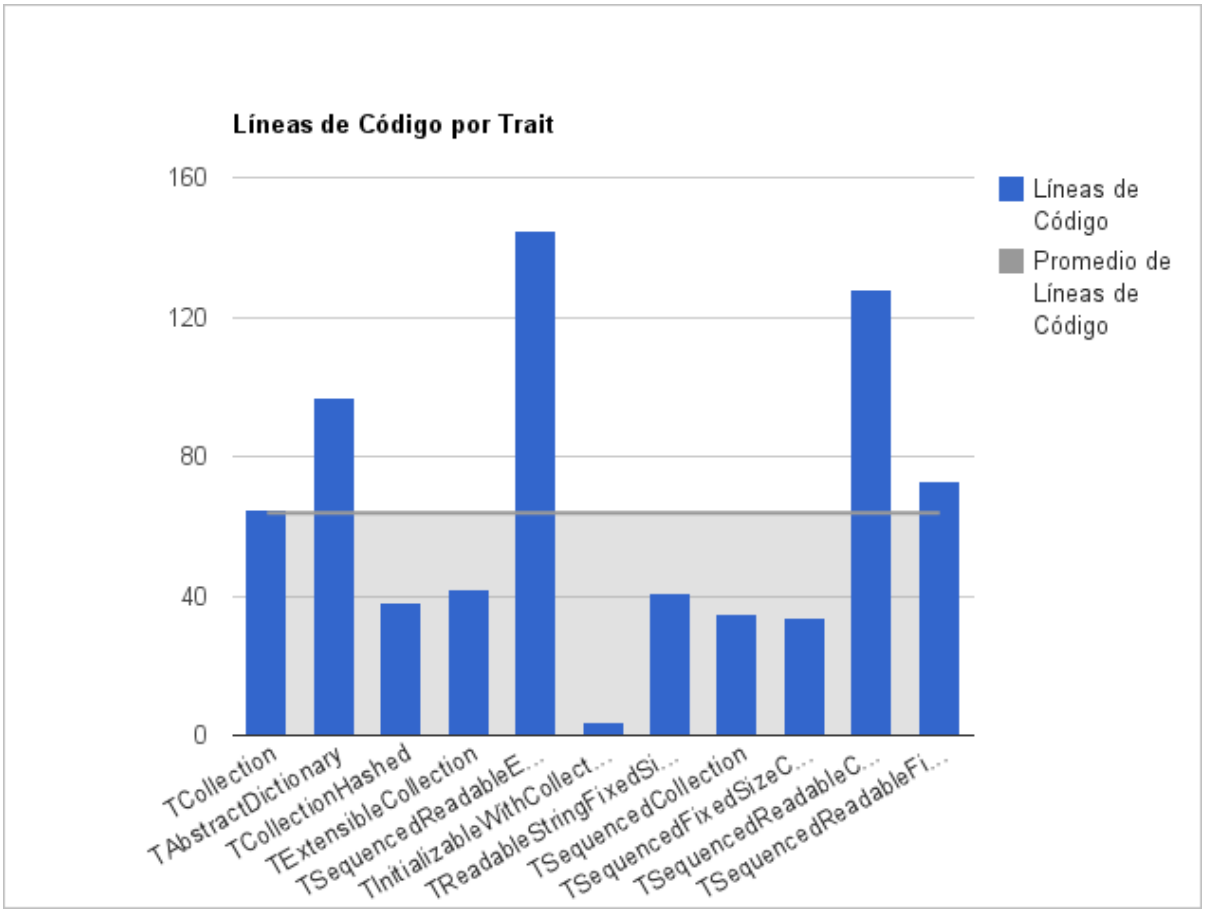

Figura 51 - Líneas de Código por Trait

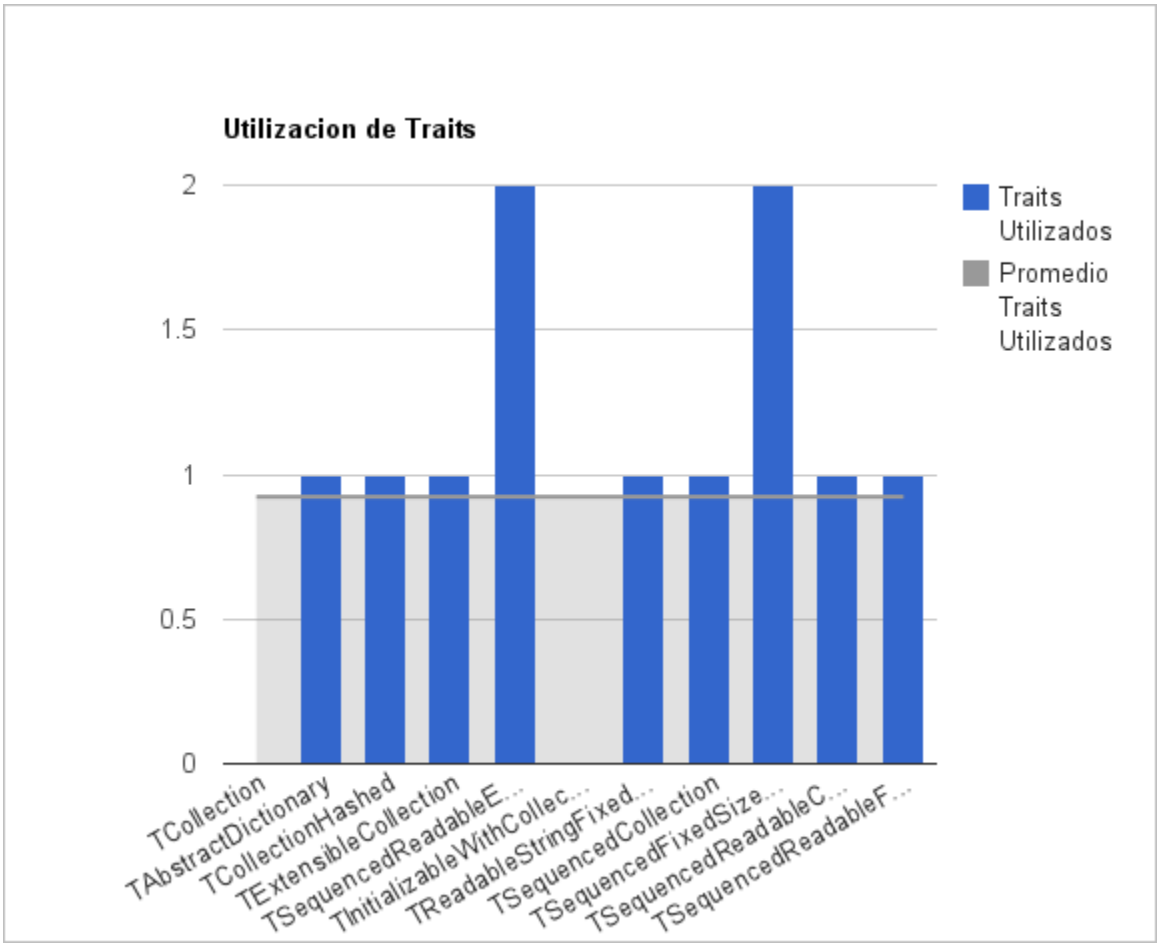

Figura 52 - Cantidad de Traits Usados por Trait

Las figuras 49, 50, 51 y 52 muestran métricas de los traits del modelo resultante.

5 Métricas del Modelo Actual de Collections de Pharo

Para poder obtener las métricas del modelo actual de Collections, fue necesario desarrollar un módulo, ya que no existían en el sistema herramientas que permitieran obtener esto de manera sencilla.
El problema principal es que es necesario contabilizar sólo los métodos del ANSI. Por otro lado, los métodos que son invocados por los del ANSI también deben ser tenidos en cuenta. Además queremos descartar métodos que no pertenecen a la jerarquía de Collection, tales como los pertenecientes al protocolo de Object.
Todo esto es necesario para poder comparar correctamente las métricas obtenidas por ambos modelos de Collection.

5.1.1 Métricas Generales

Cantidad total de clases: 15
Cantidad de clases abstractas: 4
Cantidad de clases concretas: 11
Cantidad total de métodos: 247
Cantidad total de líneas de código: 1367
Longitud de cadena más larga de subclases: 4

5.1.2 Métricas por clase

En los casos de una clase concreta con subclases, estamos contando como métodos no compartidos aquellos sobrescritos por las subclases.
Las figuras 53 y 54 muestran métricas de cantidad de métodos y cantidad de líneas de código por clase.

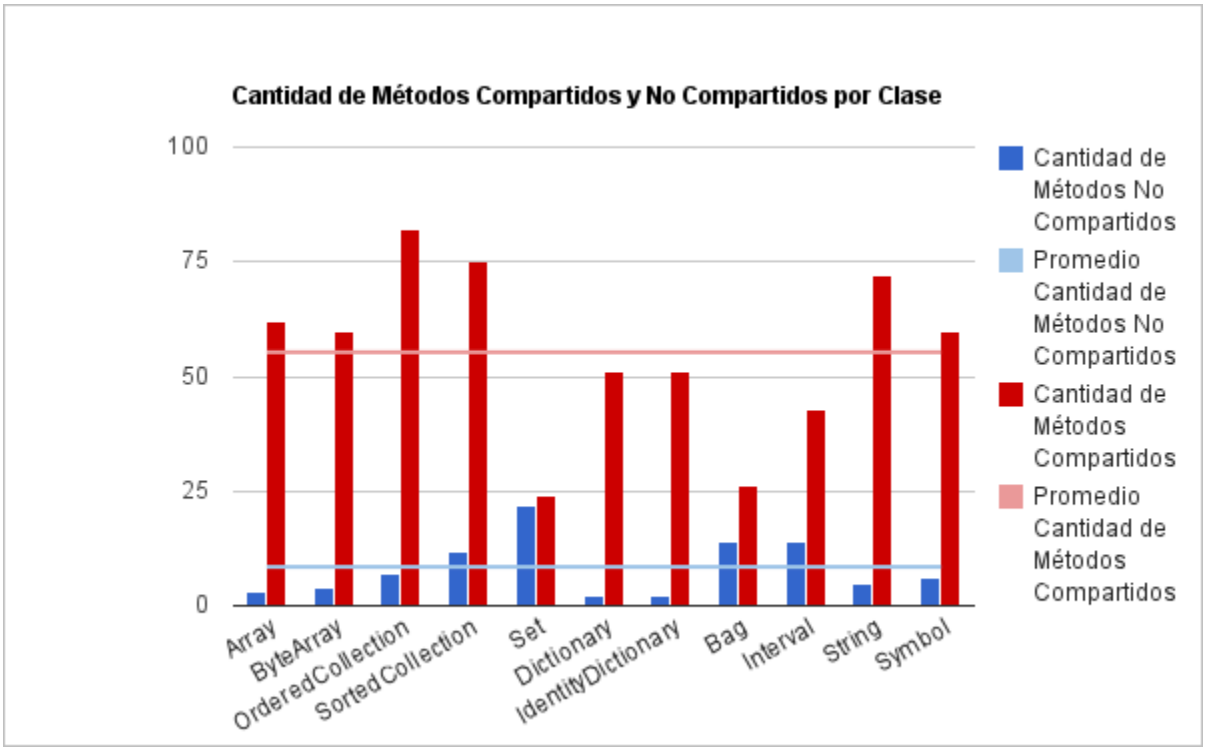

Figura 53 - Cantidad de Métodos Compartidos y No Compartidos por Clase

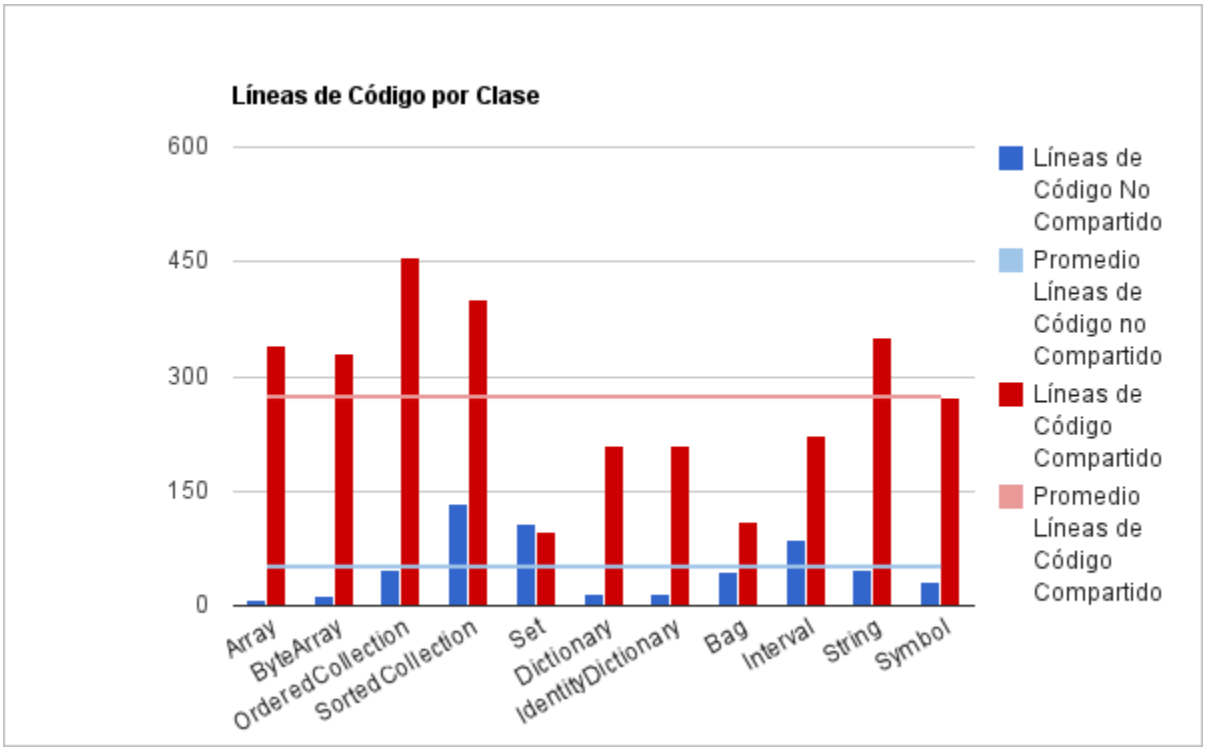

Figura 54 - Líneas de Código de Métodos Compartidos y No Compartidos por Clase

5.2 Análisis de la Jerarquía de Collections de Pharo

En esta sección realizamos un análisis buscando problemas en el modelo de Collections de Pharo.
Este análisis se limita sólo a los mensajes definidos por el ANSI para la jerarquía.

5.2.1 Métodos Cancelados

Decimos que un método es cancelado si el mismo es definido en una clase y luego es sobrescrito en una subclase lanzando una excepción.
La necesidad de cancelar un método se debe a que el método fue compartido muy arriba en la jerarquía y no todas las subclases deberían saber responder ese mensaje.

A continuación enumeramos los métodos cancelados encontrados:

- `#add:`
- `#addFirst`
- `#insertBefore:`
- `#remove:ifAbsent:`
- `#removeAll`
- `#remove`
- `#replaceFrom:to:with:startingAt:`

5.2.2 Métodos Sobrantes

Decimos que un método es sobrante si el mismo es heredado en una clase que no debería implementarlo, y ésta no lo cancela. En algunos casos el método no puede cancelarse porque es utilizado por otros métodos que sí deben ser implementados.

A continuación listamos los métodos sobrantes encontrados:

- `#addAll:`
- `#removeAll:`
- `#remove`
- `#add:after:`
- `#add:afterIndex:`
- `#add:before:`
- `#add:beforeIndex:`
- `#addAllFirst:`
- `#addAllLast:`
- `#addFirst:`
- `#addLast:`
- `#withAll:`

- `#replaceFrom:to:with:`
- `#atAll:put:`
- `#at:put:`
- `#replaceFrom:to:with:startingAt:`

5.2.3 Métodos Duplicados

Decimos que un método es duplicado si existe otro que se comporta de igual manera. Esto quiere decir, que tienen algoritmos equivalentes.
Dividimos los métodos duplicados en categorías según la dificultad para removerlos:

5.2.3.1 Métodos de Fácil Remoción

Son aquellos métodos duplicados que pueden ser simplemente removidos, sin la necesidad de realizar un refactoring.
Muchos de los métodos encontrados en esta categoría parecen ser motivo de un refactoring en la jerarquía. La versión de Pharo utilizada es la 1.1, y entre los cambios de esta figura un cambio en la jerarquía donde Dictionary y Set pasan a ser subclases de HashedCollection en vez de Set subclase de Dictionary. De hecho muchos de los métodos duplicados en esta categoría fueron removidos en versiones las versiones posteriores de Pharo.

A continuación listamos los métodos encontrados:

- Los mensajes privados `#findElementOrNil #atNewIndex:put: #fullCheck` están definidos en HashedCollection y Set con la misma implementación. Estos mensajes son utilizados por muchos métodos que corresponden al protocolo ANSI.
- `#occurrencesOf:` es definido en Dictionary con la misma implementación que en Collection
- `#asSet` es definido en Dictionary con la misma implementación que en Collection.
- `#do:separatedBy:` es definido en SequenceableCollection con un algoritmo equivalente al de Collection.
- `#includes:` definido en Dictionary con la misma implementación que en Collection.
- `#size` definido en Set con la misma implementación que en HashedCollection
- `#first` definido en Interval con una implementación equivalente a la de SequenceableCollection.
- `#last` definido en Interval con una implementación equivalente a la de SequenceableCollection.
- `#reverseDo:` definido en Interval con una implementación equivalente a la de SequenceableCollection.

5.2.3.2 Métodos que Requieren una Refactorización

En este caso, los métodos duplicados requieren de un refactoring para poder ser removidos.

A continuación listamos los métodos encontrados:
`#collect:` en SortedCollection podría utilizar la implementación de OrderedCollection si la inicialización de la colección devuelta se hiciera a través de un mensaje nuevo como `#emptyToCollect`
`#collect` en Set podría utilizar la implementación de Collection. La diferencia es que en Set se inicializa la colección a devolver con capacidad para el tamaño de la colección recibida como parámetro. Si en Collection se definiera un mensaje para inicializar la colección que se devuelve, Set podría sobrescribir este mensaje de inicialización.
`#select:` definido en Set es equivalente a `#select:` definido en Collection. La única diferencia es que Set utiliza el mensaje `#copyEmpty` para inicializar la colección a devolver, y de esta manera permite a sus subclases sobrescribir la inicialización. Esta implementación podría moverse a Collection, al igual que `#copyEmpty`.

- `#includes:` definido en SequenceableCollection con un algoritmo equivalente al de Collection, excepto para Interval y String.
  Se podría sobrescribir el método en Interval y String en vez de hacerlo en SequenceableCollection. Otra posibilidad es ver si el método en Collection es realmente necesario.

5.2.3.3 Métodos no Removibles

Definimos como métodos no removibles a aquellos para los cuales en principio no parece haber una refactorización que permita eliminarlos.
Para analizar la eliminación de este tipo de métodos habría que considerar una refactorización de la estructura de la jerarquía, lo cual está fuera del alcance de esta tesis.

A continuación listamos los métodos encontrados dentro de esta categoría:
`#collect:` definido en OrderedCollection es equivalente al `#collect:` de Collection. El problema es que SequenceableCollection redefine el método `#collect:` para que este sea compatible con sus subclases que no implementan `#add:`.
El método definido en SequenceableCollection debe luego ser sobrescrito por OrderedCollection, ya que no son compatibles.
`#select:` definido en OrderedCollection es equivalente al `#select:` de Collection. La única diferencia entre ambos es que el de OrderedCollection utiliza el mensaje `#copyEmpty` para inicializar la lista retornada, y de esta manera posibilita a las subclases a sobrescribir el mensaje de inicialización. Así es como SortedCollection sobrescribe este mensaje. Sin embargo la implementación de OrderedCollection podría utilizarse en Collection y aun así OrderedCollection debería sobrescribir la implementación de SequenceableCollection duplicando el código.

6 Comparación de Modelos

En esta sección se contrastarán las métricas obtenidas de ambos modelos.

6.1 Comparación de Métodos

Las figuras 55 y 56, muestran una comparación de la cantidad de métodos compartidos y no compartidos por clase de ambos modelos.
Podemos ver en la Figura 55 que el nuevo modelo en promedio tiene mayor cantidad de métodos no compartidos. Es necesario tener en cuenta que en esta figura se están contabilizando los métodos de acceso a variables del nuevo modelo. Los métodos de acceso son utilizados por los traits para parametrizar los métodos que estos proveen.

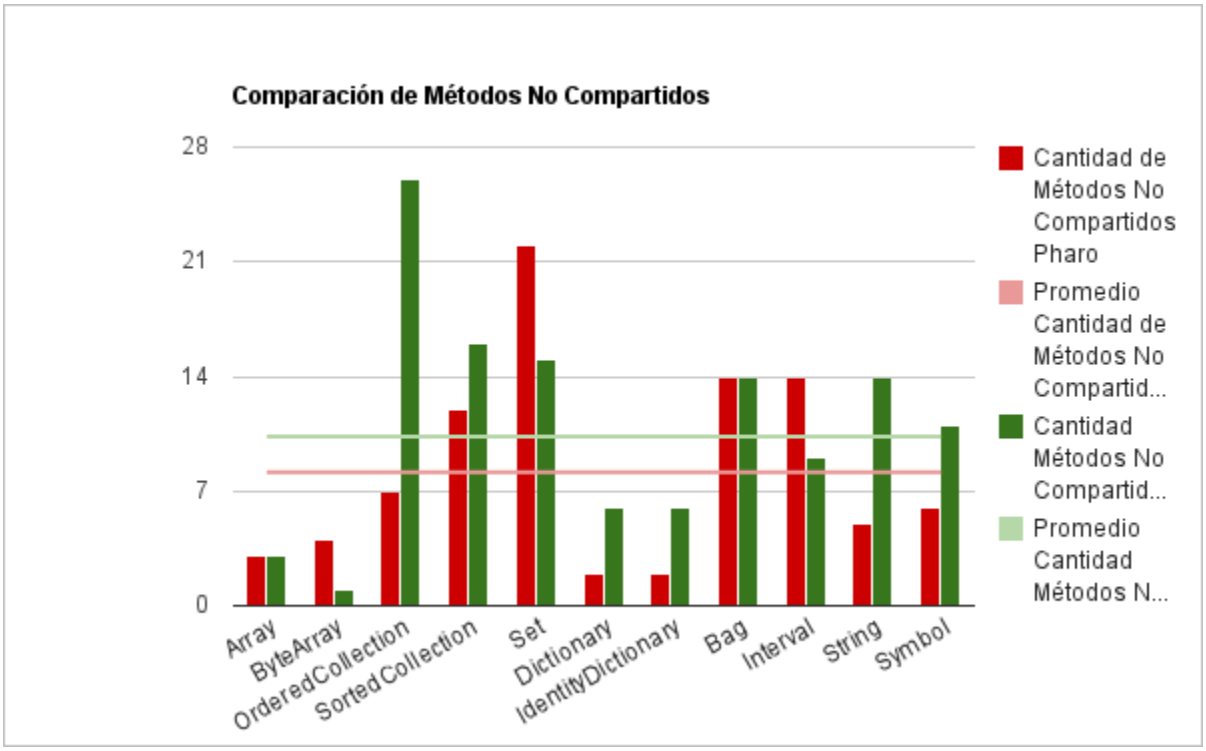

Figura 55 - Comparación de Métodos no Compartidos

Podemos ver en la Figura 56 que la diferencia en cantidad de métodos compartidos por ambos modelos en promedio es mínima. Más adelante se analizarán en detalle estos resultados.

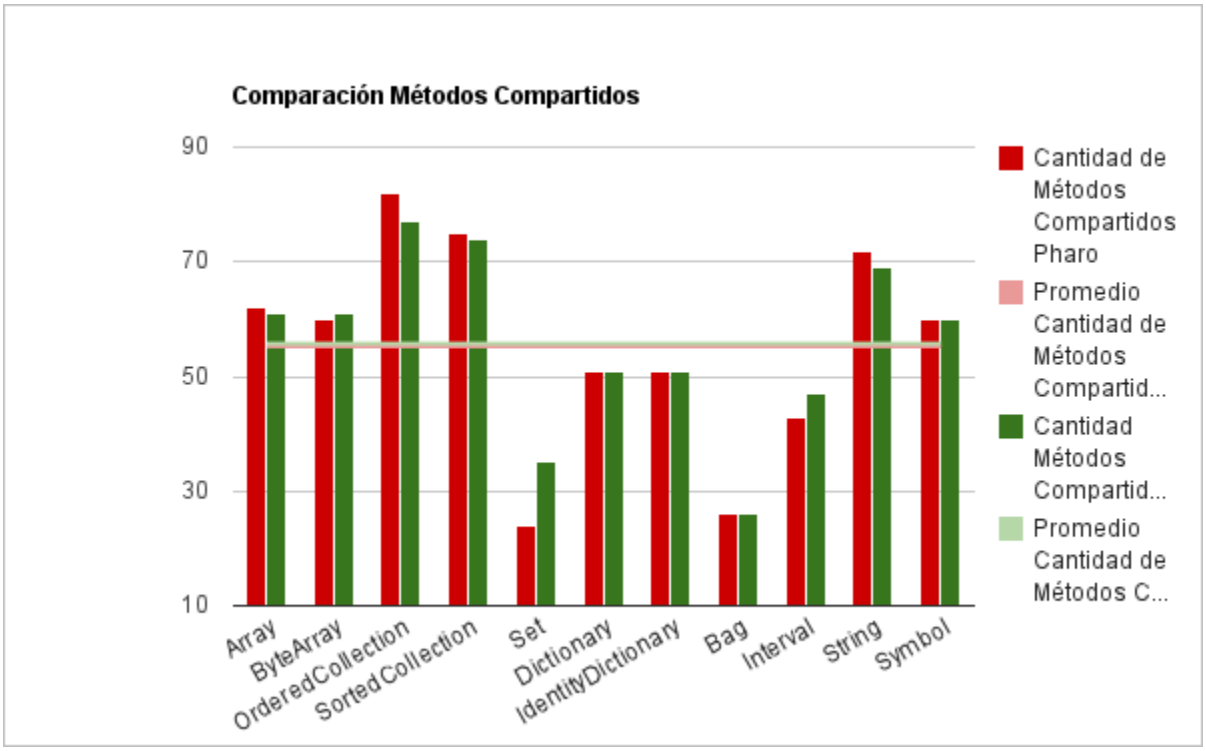

Figura 56 - Comparación de Métodos Compartidos

6.2 Comparación de Líneas de Código

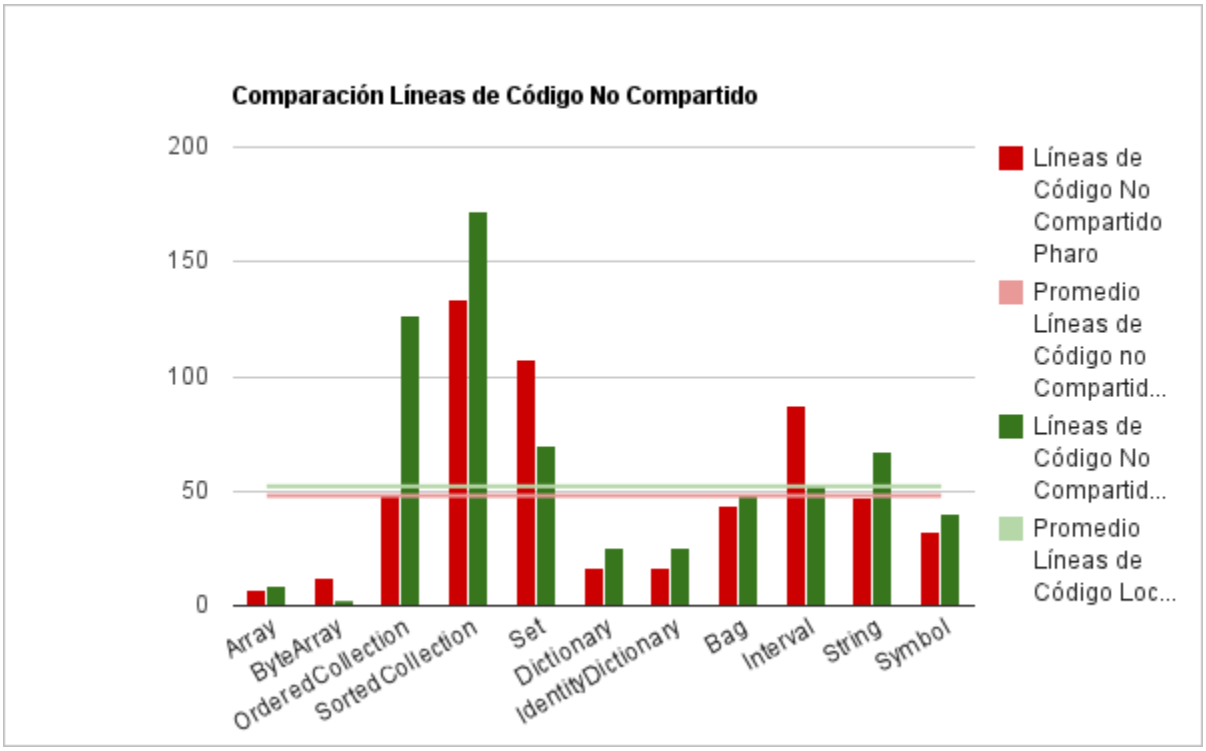

Figura 57 - Comparación de Líneas de Código No Compartido

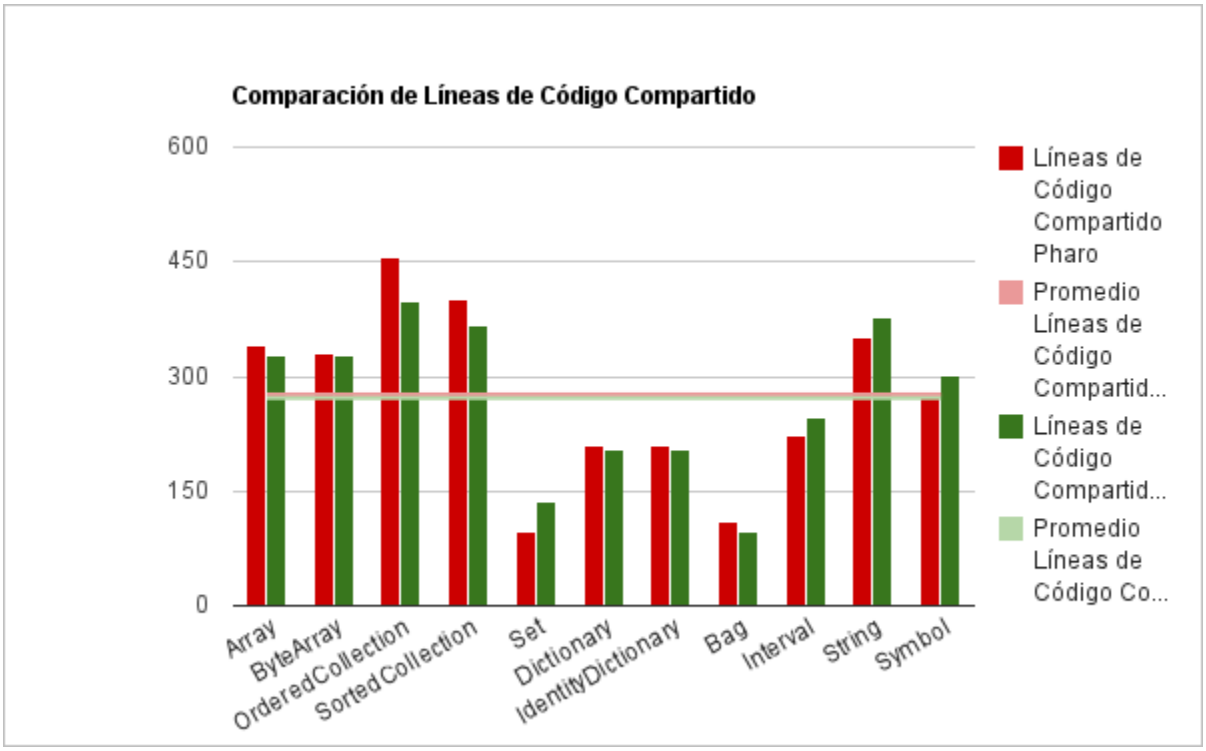

Figura 58 - Comparación de Líneas de Código Compartido

Las figuras 57 y 58, muestran la comparación de cantidad de líneas de código compartido y no compartido de ambos modelos.
Podemos ver que la diferencia en cantidad de líneas de código no compartido entre ambos modelos no es tan significativa como la cantidad de métodos no compartidos. Esto se debe a que los métodos de acceso tienen 1 líneas de código únicamente.
Con respecto a la cantidad de líneas de código compartidas, vemos que al igual que en la cantidad de métodos compartidos, la diferencia entre ambos modelos es mínima.

6.3 Caso OrderedCollection

Tanto en los gráficos de líneas de código como en los de cantidad de métodos, una de las diferencias más grandes entre ambos modelos se encuentra en la clase OrderedCollection.
Si analizamos en detalle qué es lo que sucede en esta clase notamos que OrderedCollection en el modelo de Pharo tiene mayor cantidad de métodos y líneas de código compartidas porque SortedCollection hereda muchos métodos que o bien no debería heredar como por ejemplo `#addLast: #addAllLast:`, o bien son métodos privados utilizados por los métodos que no deberían ser compartidos, por lo tanto tampoco deberían ser heredados por SortedCollection.

6.4 Caso Set

Si nos centramos en el caso de Set, vemos que el nuevo modelo posee más cantidad de métodos y líneas de código compartidas que el modelo tradicional. Y por otro lado también posee menos cantidad de líneas de código y métodos no compartidos.
Analicemos qué sucede en este caso:

Set en el modelo de Pharo redefine 9 métodos heredados de HashedCollection con exactamente la misma implementación. Como se comentó anteriormente, esto parece ser un problema producto de un refactoring hecho en la jerarquía. Muchos de estos métodos duplicados fueron removidos en versiones posteriores de Pharo.

6.5 Métodos de Acceso

La utilización de traits muchas veces requiere que se utilicen métodos de acceso, conocidos también como *accessors* o *getters* y *setters*, en vez de acceder a las variables en forma directa. La utilización de estos métodos afecta a los gráficos de métodos no compartidos mostrados anteriormente. A continuación se muestra la comparación de métodos no compartidos, teniendo en cuenta y diferenciando a estos métodos de acceso.

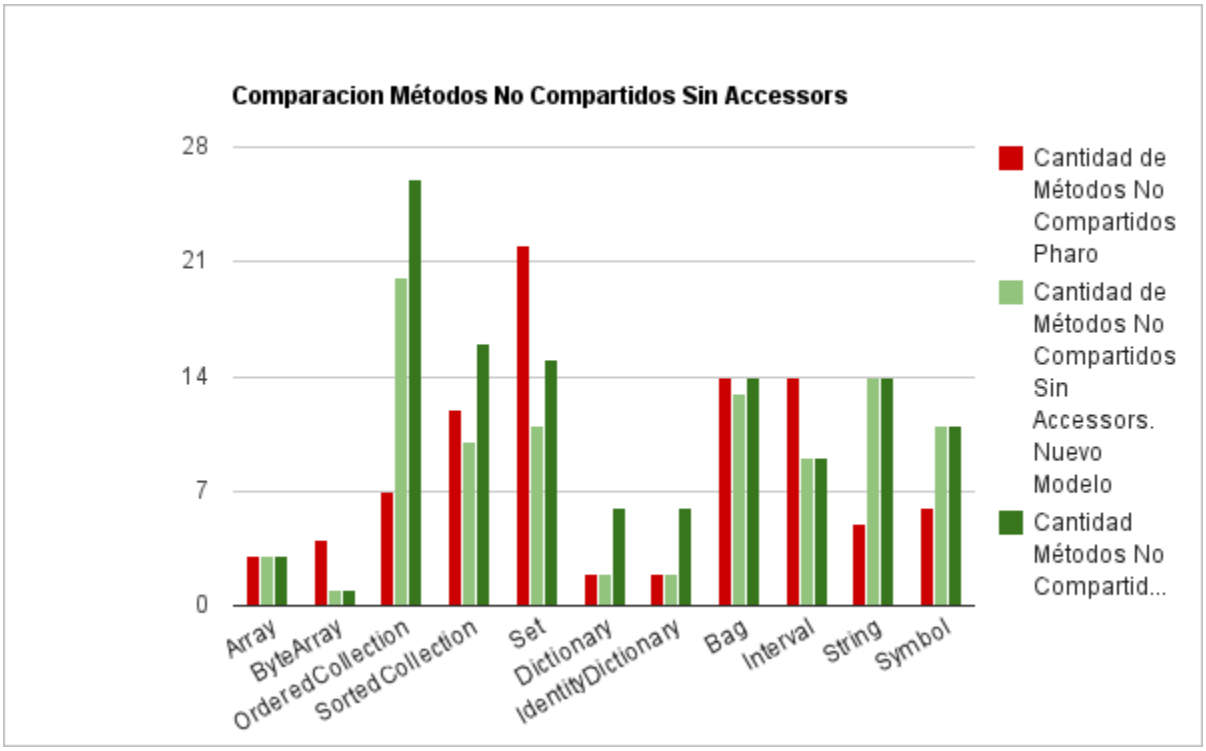

Figura 59 - Comparación de Métodos No Compartidos Sin Accessors

En la Figura 59, podemos observar que si no contabilizamos los métodos de acceso a variables, la diferencia en la cantidad de métodos no compartidos de ambos modelos se minimiza.

6.6 Comparación de Complejidad

A continuación se compararán características de ambos modelos que tengan que ver con la complejidad de los mismos. Se entiende que la complejidad de los modelos es algo difícil de medir y comparar y, además, existe una cuota de subjetividad a la hora de evaluar este tipo de característica.

Modelo de Collections con Subclasificación	Modelo de Collections sólo con Traits
Total de 15 clases	Total de 11 traits y 11 clases
4 clases abstractas	11 traits
11 clases concretas	11 clases concretas
La longitud de la cadena más larga de subclasificación es 4	La Longitud de la cadena más larga de uso de traits es 4
14 relaciones entre elementos del modelo	33 relaciones entre elementos del modelo
Dado que es un modelo de subclasificación simple, una clase tiene una única superclase	El promedio de utilización de traits por clase es de 2.36
Cantidad total de métodos 247	Cantidad total de métodos 257
Cantidad total de líneas de código 1367	Cantidad total de líneas de código 1314
Elementos del modelo: Clases, relación de subclasificación simple y objetos.	Elementos del modelo: Clases, relación de subclasificación simple, traits, relación de uso de traits y objetos.

Métodos cancelados:
`#add:`
`#addFirst`
`#insertBefore:`
`#remove:ifAbsent:`
`#removeAll`
`#remove`
`#replaceFrom:to:with:startingAt:`
	Ningún método cancelado

Métodos sobrantes:
`#addAll:`
`#removeAll:`
`#remove`
`#add:after:`
`#add:afterIndex:`
`#add:before:`
`#add:beforeIndex:`
`#addAllFirst:`
`#addAllLast:`
`#addFirst:`
`#addLast:`
`#withAll:`
`#replaceFrom:to:with:`
`#atAll:put:`
`#at:put:`
`#replaceFrom:to:with:startingAt:`
	Ningún método sobrante

Métodos duplicados: `#findElementOrNil` en Set y HashedCollection. `#atNewIndex:put:` en Set y HashedCollection. `#fullCheck` en Set y HashedCollection. `#occurrencesOf:` en Dictionary y Collection `#asSet` definido en Dictionary y Collection `#do:separatedBy:` en SequenceableCollection y Collection `#includes:` en Dictionary y Collection `#size` en Set y HashedCollection `#first` en Interval y SequenceableCollection `#last` en Interval y SequenceableCollection `#reverseDo:` en Interval y SequenceableCollection `#collect:` en SortedCollection y OrderedCollection `#collect:` en Set y Collection `#select:` en Set y Collection `#includes:` SequenceableCollection y Collection `#collect:` en OrderedCollection y	Ningún método duplicado.

Collection `#select:` en OrderedCollection y Collection	

7 Discusiones

Esta sección presenta diferentes análisis que son relevantes a los objetivos de este trabajo. Los mismos fueron realizados a partir de las métricas realizadas en las secciones anteriores.

7.1 Métodos Cancelados

El modelo actual de Collections de Pharo posee varios métodos cancelados[7]. Uno de los métodos cancelados es `#add:`. Este mensaje es implementado en Colletion como subclassResponsibility, y luego redefinido como `#shouldNotImplement` en subclases como Array.
Esto claramente es un error de diseño, ya que el modelo supone que todas las subclases de Collection deberían implementar ese mensaje cuando no es así.
Una solución simple al problema sería eliminar el método `#add:` de la clase Collection e implementarlo únicamente en las subclases que corresponda.
Por lo tanto, podemos concluir que los métodos cancelados no son un problema inherente del modelo de subclasificación, sino que son un problema de este modelo en particular.
Por su parte, el modelo utilizando traits no posee métodos cancelados. La composición de traits permite excluir métodos a la hora de utilizar un trait, por lo que no tendría sentido tener métodos cancelados.

7.2 Métodos Sobrantes

7.2.1 OrderedCollection y SortedCollection

Como vimos anteriormente en el caso de OrderedCollection, existen métodos que SortedCollection no debería heredar de OrderedCollection. Los mismos podrían ser cancelados en SortedCollection, pero esto habría que analizarlo dado que puede que esos métodos sean utilizados por SortedCollection.
De todas maneras, que una instancia de SortedCollection sepa responder el mensaje `#addFirst` y que, de hecho, lo haga sin devolver una excepción, parece un error grave en el modelo ya que puede la instancia puede quedar en estado inconsistente.

[7] Un método se dice cancelado en Smalltalk si el mismo es definido en una clase y luego es redefinido en una subclase enviándose a sí mismo el mensaje `#shouldNotImplement`, lo cual lanza una excepción.

A diferencia del caso de métodos cancelados, borrar el método de la superclase no es una posibilidad, ya que la superclase es una clase concreta y está bien que sepa responder el mensaje.
Una posibilidad es agregar una clase abstracta en el lugar de OrderedCollection, y hacer que OrderedCollection y SortedCollection compartan comportamiento a través de ella.
Aunque la solución para este problema es un tanto más compleja, podría resolverse, y parece ser más un problema de este caso en particular, que un problema del paradigma de subclasificación.
Por otro lado en el modelo utilizando traits, este problema no se encuentra presente ya que OrderedCollection y SortedCollection comparten comportamiento a través de un trait, y eso hace que se pueda optar por no poner el método en el trait, ya que no es compartido, o ponerlo y luego excluirlo en la composición. En el caso de este modelo, el mismo no posee métodos excluidos.

7.2.2 Método #remove:

Este es otro caso de métodos sobrantes. Analizaremos en particular la implementación del método `#remove:`. Aunque analizamos el caso del método `#remove:`, el mismo problema ocurre con otros métodos.
Para comprender el problema necesitamos primero conocer la jerarquía de clases de Collection en Pharo por un lado, y tener en cuenta las relaciones entre los protocolos del ANSI de Collection por otro.
El método `#remove:` pertenece al protocolo ExtensibleCollection del ANSI. Y esto indica, según el ANSI, que el mensaje debe ser implementado por los protocolos: Bag, Set, OrderedCollection y SortedCollection. Ahora ubiquemos estas mismas clases en la jerarquía de Collections de Pharo. La implementación del mensaje puede ser compartida, y la única clase común entre todas esas clases es Collection. Esto significa que si se quiere compartir la implementación del mensaje, tiene que implementarse en esa clase.
El problema de implementar el mensaje `#remove:` en la clase Collection es que todas las subclases lo heredan, y eso es algo que no se quiere hacer. Por ejemplo, es un error que objetos instancia de Interval sepan responder ese mensaje.
A diferencia de los casos anteriores como métodos cancelados y los métodos sobrantes de OrderedCollection y SortedCollection, este problema no tiene una solución correcta en el paradigma de subclasificación simple.

Para entender por qué no existe una solución en el modelo de subclasificación simple, veamos una característica de un conjunto de implementaciones de protocolos que hace que no exista una buena representación en el paradigma.
Supongamos que necesitamos representar la situación que muestra la Figura 60.

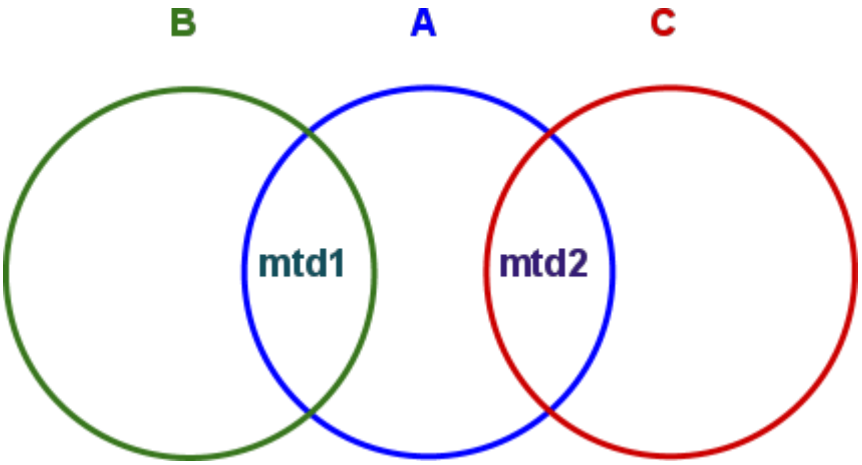

Figura 60 - Escenario de Métodos Compartidos con Diferentes Clases

Veamos intuitivamente, que esta condición hace que sea imposible la representación mediante subclasificación simple.
Si A debe tener a mtd1 y mtd2, creamos una clase A con ambos métodos, como muestra la Figura 61.

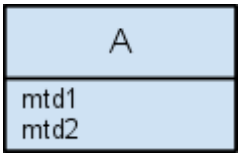

Figura 61 - Clase A con mtd1 y mtd2

Ahora si queremos representar a B creamos una clase B, pero mtd1 necesitamos compartirlo con A. Si B subclasifica de A, también hereda mtd2 lo cual no corresponde. Lo que podemos hacer es mover mtd1 a B y hacer que A subclasifique de B, como muestra la Figura 62. De esta manera A y B quedan correctamente representados.

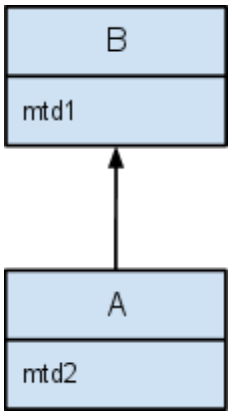

Figura 62 - Clase B con mtd1 y como superclase de A

El problema surge cuando queremos representar a C, dado que C necesita compartir con A el método mtd2, pero A no puede obtener mtd2 de una superclase porque ya hereda de B, y si C subclasifica de A no solo hereda mtd2 sino también mtd1, lo cuál no corresponde.
Esta situación se ilustra en la Figura 63.

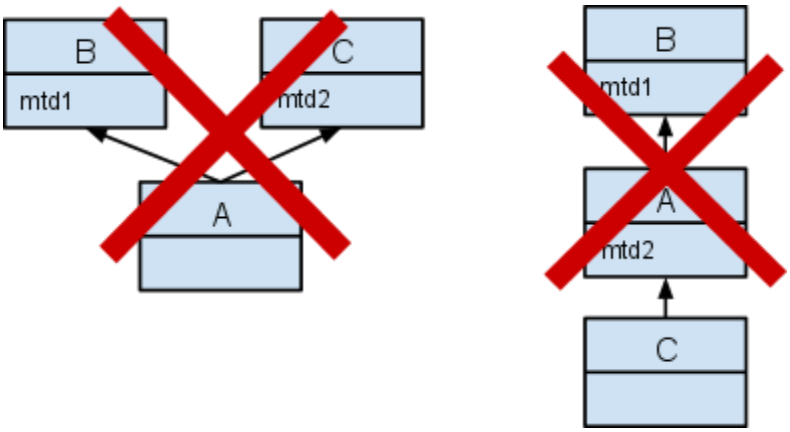

Figura 63 - Problemas para Representar la Clase C

Análogamente podemos ver que pasa lo mismo si hacemos que A subclasifique de C.

La demostración formal de este teorema, se encuentra en la sección Demostración del Teorema de Representación de Subclasificación Simple.

Ahora veamos, en la Figura 64, que el problema que tenemos cumple con esta misma condición:

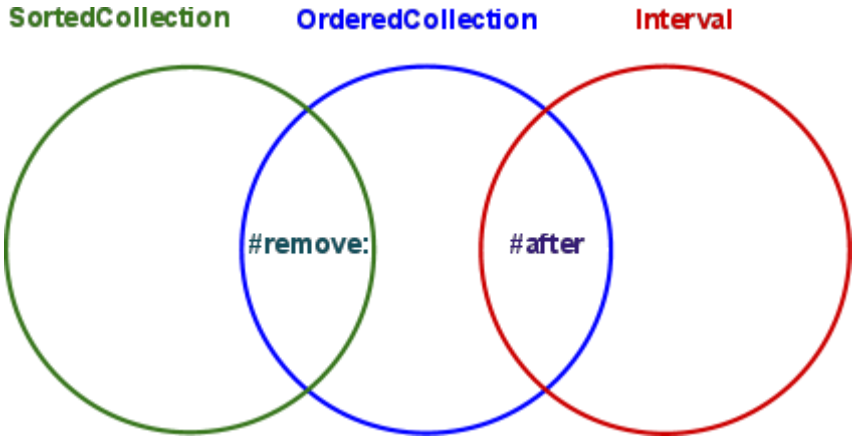

Figura 64 - Escenario de Métodos Compartidos con Diferentes Clases en Collections

Por lo tanto podemos concluir que no existe forma de compartir la implementación de todos los mensajes en las clases de la manera presentada utilizando un modelo de subclasificación simple. Las soluciones posibles dentro del paradigma son, o bien repetir código o hacer que clases hereden métodos que no les corresponden.

En una implementación que utiliza traits para compartir comportamiento no tenemos este problema de representación.
La relación *usa* que existe entre clases y traits, y entre traits y traits, es diferente a la relación de subclasificación. La diferencia radica en la cardinalidad de la relación, mientras que en la subclasificación simple la relación es 1:n, la relación *usa* es n:m.
Puesto de otra forma: en herencia simple, una clase puede ser subclase de una y solo una clase, y una clase puede tener “n” subclases. Utilizando traits, una clase puede utilizar “n” traits, y los traits pueden ser utilizados por “m” clases. Y cuando la relación es entre traits: un trait puede utilizar “n” traits y un trait puede ser utilizado por “m” traits.
Al no tener las restricciones del modelo de herencia simple, utilizando traits podemos representar sin problemas el problema antes planteado. La Figura 65 muestra una solución al problema utilizando traits.

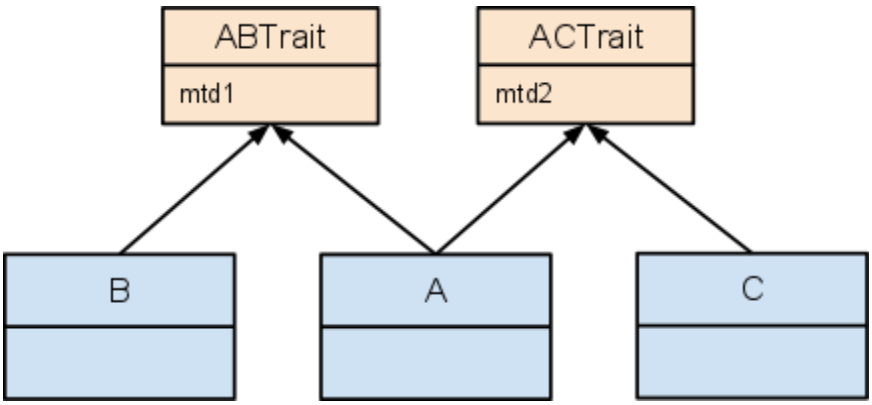

Figura 65 - Solución a Métodos Compartidos con Diferentes Clases Usando Traits

Podemos decir entonces que el modelo de traits puede representar de forma correcta ciertos problemas con los cuales el modelo de subclasificación simple tiene dificultades para representar.
Las ventajas de no tener código repetido en un modelo son claras. Pero no tener métodos sobrantes también es un gran beneficio. Heredar métodos que no corresponden crea un modelo que representa de manera errónea la realidad y puede generar un concepto erróneo de lo que se está representando. En muchos casos, el responder mensajes que no corresponden puede generar errores e inconsistencias en el modelo.
Por lo tanto, que con traits se pueda resolver este problema de forma prolija no es un dato menor.

7.3 Herencia de Variables

Una de las diferencias entre el modelo de Subclasificación y el de traits es que en el modelo de subclasificación se heredan las variables de una clase, mientras que mediante traits esto no es posible.

Esto hace que los traits necesiten de métodos de acceso a las variables de instancia, y que los métodos compartidos, utilicen esos métodos de acceso.
El hecho de que no se puedan heredar las variables de instancia puede ser un aspecto positivo, ya que en muchos casos la subclasificación es mal utilizada por este motivo.
Muchas veces se ven jerarquías de clases que no comparten comportamiento sino que comparten únicamente variables.
El compartir variables puede resultar más cómodo desde el punto de vista del desarrollo, pero, compartir únicamente variables, puede generar un modelo que no represente de manera correcta el problema, es decir, un modelo sensible ante cambios.

7.4 Modelado Utilizando Únicamente Traits

En esta sección se describirán las experiencias obtenidas durante el desarrollo del nuevo modelo de Collections.

Al momento de crear un modelo se está representando la realidad utilizando elementos y reglas del paradigma que elegimos. Al cambiar la forma de compartir comportamientos, estamos cambiando las herramientas que tenemos para modelar, y también las reglas que utilizamos para la construcción de nuestro modelo.

A la hora de crear un modelo utilizando nuevas herramientas, las reglas o axiomas de construcción no están tan claras. Algo importante de un modelo es la consistencia de representación. Con esto nos referimos a que entes similares o iguales deben ser representados de manera similar o igual en nuestro modelo. Y este fue el punto de partida para la construcción del nuevo modelo.
Dado que seguimos utilizando objetos y clases, ya sabíamos que los objetos representarían colecciones y las clases los comportamientos de estas. La pregunta era, qué representarían los traits.
Al igual que las clases abstractas, los traits representan comportamiento común entre clases. Y ya que estamos representando el protocolo ANSI de Collections, utilizamos la distribución de mensajes en protocolos que define el ANSI para crear los traits de nuestro sistema. De esta manera, cada protocolo abstracto sería representado por un traits. Luego nos dimos cuenta que en realidad con un trait estamos representando implementaciones de protocolos, ya que un protocolo puede tener más de una implementación. En definitiva los protocolos abstractos son comportamiento común, pero además los mensajes contenidos en el mismo tienen que ver esa característica común que se está modelando.
La ventaja de utilizar traits para representar los protocolos es que, los mensajes de estos últimos ya fueron divididos de manera correcta, de forma tal que cada clase implementa sólo los mensajes que debe. El resultado de esto es que no fue necesario realizar exclusiones de métodos a la hora de componer traits.

Por otro, lado la composición de traits representa correctamente la relación de cumplimiento que existe entre protocolos y esto también influyó para que utilicemos esta forma de representación.

Podemos ver entonces que la idea utilizada para representar el nuevo modelo de Collections es bastante simple. No podríamos haber hecho lo mismo en un modelo con subclasificación, ya que, como vimos anteriormente, la relación entre protocolos no puede representarse correctamente

8 Conclusiones

Esta tesis muestra la creación de un nuevo modelo de Collections utilizando únicamente traits como mecanismo para compartir comportamiento. Realiza un análisis tanto del nuevo modelo como del modelo actual de Collections que utiliza subclasificación. Los análisis se basan en las métricas realizadas, y luego se contrastan sus resultados para obtener conclusiones.

Teniendo en cuenta todo lo visto anteriormente, podemos concluir que, un modelo que utiliza traits como único mecanismo para compartir comportamiento entre objetos tiene como principal ventaja la flexibilidad. Con esto nos referimos al hecho de poder representar de manera correcta situaciones como la del método `#remove:`.

Por otro lado, es realmente difícil asegurar que un modelo posee mayor complejidad que otro, dado que esto es algo bastante subjetivo. Pero podemos decir que, con respecto a la experiencia realizada, la complejidad en cuanto al modelado y el desarrollo no fue significativamente diferente a la encontrada en desarrollos tradicionales que utilizan subclasificación. Además, muchos refactorings fueron bastante simples por la flexibilidad que brindan los traits. Esto es algo que facilita el desarrollo.
Si bien el desarrollo no mostró mayor complejidad, según las métricas obtenidas, podemos ver que el modelo que utiliza únicamente traits posee mayor cantidad de abstracciones, mayor cantidad de relaciones entre elementos del modelo, y además, estamos agregando el concepto de trait al modelado. Esto es un factor a tener en cuenta a la hora de evaluar la complejidad del modelo.

Concretamente con respecto al modelo de Collections obtenido, el mismo no posee métodos cancelados, ni sobrantes, ni duplicados. Esta es una gran mejora con respecto al modelo tradicional de Collections. El tener métodos cancelados, sobrantes o duplicados son síntomas de lo mismo: problemas de representación. Decimos que son síntomas de lo mismo porque ante el problema de representación con subclasificación simple siempre tenemos dos alternativas: compartir código innecesariamente, o duplicarlo.

Podemos ver entonces que existe una ventaja con respecto a la flexibilidad, pero una desventaja con respecto a la complejidad. Teniendo en cuenta esto último, consideramos útil el uso de este tipo de modelos en dominios que requieran de mayor flexibilidad.

Consideramos que, para minimizar la complejidad de este tipo de modelos, es importante el estudio y desarrollo de herramientas que faciliten su implementación. Así como también es importante la aplicación de diseños utilizando solo traits a diferentes dominios, y a partir de esto poder crear reglas o heurísticas de diseño, que produzcan un modelo más simple y compresible. Por reglas o heurísticas de diseño nos referimos por ejemplo a patrones de diseño[6] que utilicen únicamente traits, reglas que determinen cuándo es correcto utilizar la

exclusión de métodos en la composición de traits, o que indiquen cuándo corresponde agrupar varios traits en uno solo.

Respecto del modelado utilizando únicamente traits como mecanismo para compartir comportamiento, encontramos que los traits, al igual que las clases representan comportamiento de objetos, con la diferencia de que las clases además tienen la responsabilidad de la creación de instancias. Aunque clases y traits representan comportamiento, difieren en cómo lo comparten.
Teniendo en cuenta esto consideramos que un trait debe representar comportamiento homogéneo, de otra manera es probable que cada vez que se use el trait deban excluirse métodos. Consideramos la exclusión sistemática de métodos un mecanismo propenso a errores, dado que puede generar inconsistencias. Por inconsistencias nos referimos a casos en los cuales no se hayan excluido métodos que utilizan los excluidos. En otros casos puede que no se produzcan inconsistencias, pero se pueden generar métodos sobrantes, simplemente olvidando excluir ciertos métodos, lo cual produce un modelo confuso.
Por todo esto, concluimos que la exclusión regular de métodos en las composiciones de un trait, sugiere un reagrupamiento de los métodos del mismo en traits más homogéneos. Sin embargo consideramos la exclusión de métodos una herramienta útil en el modelado de casos excepcionales. En el modelo generado de Collections no se ha utilizado la exclusión de métodos.

9 Trabajo Futuro

Esta sección pretende exponer temas relacionados y a la vez fuera del alcance de de esta tesis.

- Estudiar cuestiones relacionadas con la performance de modelos que sólo utilizan traits para compartir comportamiento:
    - Analizar el impacto de acceso a variables de instancia a través de métodos
    - Analizar el impacto en el algoritmo de method lookup
    - Obtener estadísticas de performance

- Experimentar con otras jerarquías. Se podría tener en cuenta la profundidad de las mismas.

- Analizar si es posible realizar un algoritmo para transformar jerarquías existentes a un modelo de que utiliza solo traits.

- Crear herramientas de análisis estático que soporte un modelo que sólo utiliza traits.

- Analizar la posibilidad de refactorizar el modelo de traits creado, de manera de reducir la complejidad del mismo.

- Aplicar traits sin subclasificación en dominios que pueden resolverse mediante patrones de diseño[6] y a partir de esto obtener reglas o heurísticas de diseño como por ejemplo:
    - Patrones de diseño que utilizan únicamente traits para compartir comportamiento.
    - Reglas o heurísticas que permitan determinar cuándo es correcto excluir métodos en la composición de traits.
    - Reglas o heurísticas que indiquen cuándo corresponde agrupar distintos traits en uno.

Bibliografía

- [1] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Andrew Black, ***"Traits: Composable Units of Behavior,"*** Proceedings of European Conference on Object-Oriented Programming (ECOOP'03), LNCS, vol. 2743, Springer Verlag, July 2003, pp. 248-274.

- [2] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts and Andrew Black, **"*Traits: A Mechanism for fine-grained Reuse,"*** *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 2, March 2006, pp. 331-388.

- [3] Andrew P. Black, Nathanael Schärli and Stéphane Ducasse, ***"Applying Traits to the Smalltalk Collection Hierarchy,"*** *Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, vol. 38, October 2003, pp. 47-64.

- [4] Damien Cassou, Stéphane Ducasse, and Roel Wuyts, ***"Redesigning with Traits: the Nile Stream trait-based Library"***.

- [5] Nathanael Schärli, ***"Traits — Composing Classes from Behavioral Building Blocks,"*** Ph.D. thesis, University of Berne, February 2005.

- [6] Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. ***Design Patterns.Elements of Reusable Object-Oriented Software*****.** Reading, MA, Addison Wesley, 1995.

- [7] Squeak: http://www.squeak.org

- [8] Dahl, O.-J. and K. Nygaard (1967). **-SIMULA- A language for Programming and Description of Discrete Event Systems**. Oslo 3, Norway, Norwegian Computing Center, Forskningveien 1B, 5th edition, September 1967, 124 pages.

- [9] Adele Goldberg and David Robson. ***"Smalltalk-80: The Language and its Implementation"***. Addison-Wesley, 1983, ISBN 0-201-11371-6.

- [10] David Ungar, Randall B. Smith. ***"Self: The Power of Simplicity".*** 1987, ISSN:0362-1340, Páginas: 227 – 242.

- [11] Alan Snyder. ***"Encapsulation and inheritance in object-oriented programming languages"***. In Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, pages 38–45, November 1986.

- [12] Gilad Bracha, William R. Cook: **"*Mixin-based Inheritance*".** September 1990. ACM SIGPLAN Notices , Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications OOPSLA/ECOOP '90, Volume 25 Issue 10

- [13] Pharo: http://pharo-project.org/

- [14] MIT Licence: http://www.opensource.org/licenses/mit-license.php

- [15] Acciaresi Claudio, Butarelli Nicolás Martín. ***"Reingeniería de Jerarquías Polimorficas Utilizando Traits"*** FCEyN UBA, diciembre de 2008

- [16] Protocolo ANSI para Smalltalk revisión 1.9 http://wiki.squeak.org/squeak/uploads/172/standard_v1_9-indexed.pdf

- [17] Test Driven Development - Beck, K. **Test-Driven Development by Example**, Addison Wesley, 2003

- [18] Alan M Turing, **"On Computable Numbers, with an Application to the Entscheidungsproblem"**, May 1936

Anexo A

Demostración del Teorema de Representación de Subclasificación Simple

A.1 Definición del Teorema

Siendo P un conjunto de implementaciones de protocolos[8].

$$\forall\, P(NSSR(P) \Rightarrow \neg\, \exists\, R\, GSSR(R, P))$$

Donde NSSR (No Simple Subclassing Representation) es la propiedad de un conjunto de implementaciones de protocolos de no poder ser representado con subclasificación simple.

$$NSSR(P) \Leftrightarrow$$
$$\exists\, pi, pj, pk\ (pi \in P\ \wedge pj \in P \wedge pk \in P$$
$$\wedge\, (methods(pi) \cap methods(pj)) - methods(pk) \neq \oslash$$
$$\wedge\, (methods(pi) \cap methods(pk)) - methods(pj) \neq \oslash)$$

Si P cumple con esta propiedad, quiere decir que tiene una implementación de protocolo que debe compartir diferentes métodos, con otras dos implementaciones de protocolos diferentes.

GSSR(R, P) (Good Simple Subclassing Representation) es un predicado que dice si R es una buena representación de P.
Dado que R es un modelo del paradigma de subclasificación simple, el mismo representa a P de manera correcta si:
Cada p∈P es representado por una clase en R
Si c es una clase que pertenece a R y representa a p que pertenece a P, entonces methods(c)=methods(p).
No existen en R métodos duplicados

[8]Una implementación de protocolo es un conjunto de métodos

Formalmente:

$$GSSR(R,P) \Leftrightarrow \forall p\, \exists c(p \in P \wedge c \in classes(R) \Rightarrow methods(c) = methods(p)))$$
$$\wedge \neg \exists m,c1,c2\,(c1 \in classes(R) \wedge c2 \in classes(R) \wedge c1 \neq c2$$
$$\wedge m \in methods(c1) \wedge m \in methods(c2))$$
$$\wedge \forall c(c \in R \Rightarrow \#(superclasses(c)) < 2)$$

A.2 Demostración del Teorema

$$\forall P(NSSR(P) \Rightarrow \neg \exists R\, GSSR(R,P))$$

Por absurdo:
$$\exists P(\exists R\, GSSR(R,P) \wedge NSSR(P))$$

Por NSSR(P) sabemos que:

$$\exists pi, pj, pk\,(pi \in P \wedge pj \in P \wedge pk \in P$$
$$\wedge (methods(pi) \cap methods(pj)) - methods(pk) \neq \oslash$$
$$\wedge (methods(pi) \cap methods(pk)) - methods(pj) \neq \oslash)$$

De acuerdo a esto último, sean entonces m1 y m2 métodos tales que:
$$m1 \in pi \wedge m1 \in pj \wedge m1 \notin pk$$
$$m2 \in pi \wedge m2 \in pk \wedge m1 \notin pj$$

Por GSSR(R,P) sabemos que:

1. $\forall p\, \exists c(p \in P \wedge c \in classes(R) \Rightarrow methods(c) = methods(p)))$
2. $\neg \exists m,c1,c2\,(c1 \in classes(R) \wedge c2 \in classes(R) \wedge c1 \neq c2$
   $\wedge m \in methods(c1) \wedge m \in methods(c2))$
3. $\forall c(c \in R \Rightarrow \#(superclasses(c)) < 2)$

Por 2, sabemos que m1 y m2 deben pertenecer a dos clases diferentes de R, c1 y c2.
Por 1, sabemos que en R existen clases ci, cj, y ck que tienen los métodos de pi, pj y pk respectivamente.
Dado que m1 y m2 pertenecen a pi, para que ci tenga los métodos de pi, existen las siguientes posibilidades:
Que ci subclasifique de c1 y c2, lo cual es absurdo, ya que contradice el puto 3 de GSSR

Que ci sea igual a c1 y c1 subclasifique de c2.
Ahora ck debe ser igual a c2 para cumplir con el punto 1 de GSSR.
Pero cj no puede ser igual a c1, ni a c2 porque contradice el punto 1 de GSSP. Tampoco puede subclasificar de c1, ya que hereda de c2 y contradice el punto 1 de GSSP.
Que ci sea igual a c2 y c2 subclasifique de c1. Este caso es análogo al anterior dado que tenemos el mismo problema pero invertido.

Hemos llegado al absurdo en todos los casos, por lo tanto esto demuestra:
$\forall\, P(\neg\, \exists\, R\, GSSR(R, P) \Leftarrow NSSR(P))$

Anexo B

Protocolo Collection del ANSI

B.1 Introducción

En esta sección se mencionarán aspectos importantes del protocolo Collection del ANSI que tengan relevancia para el desarrollo del nuevo modelo.
Para detalles sobre el protocolo se debe consultar la documentación del mismo [16].

B.2 Relaciones entre Protocolos de Collection del ANSI

La Figura 66 muestra el cuadro que relaciona los protocolos de Collection.
Los nombres de los protocolos en mayúscula indican que debe existir una clase concreta en el sistema que lo implementa.

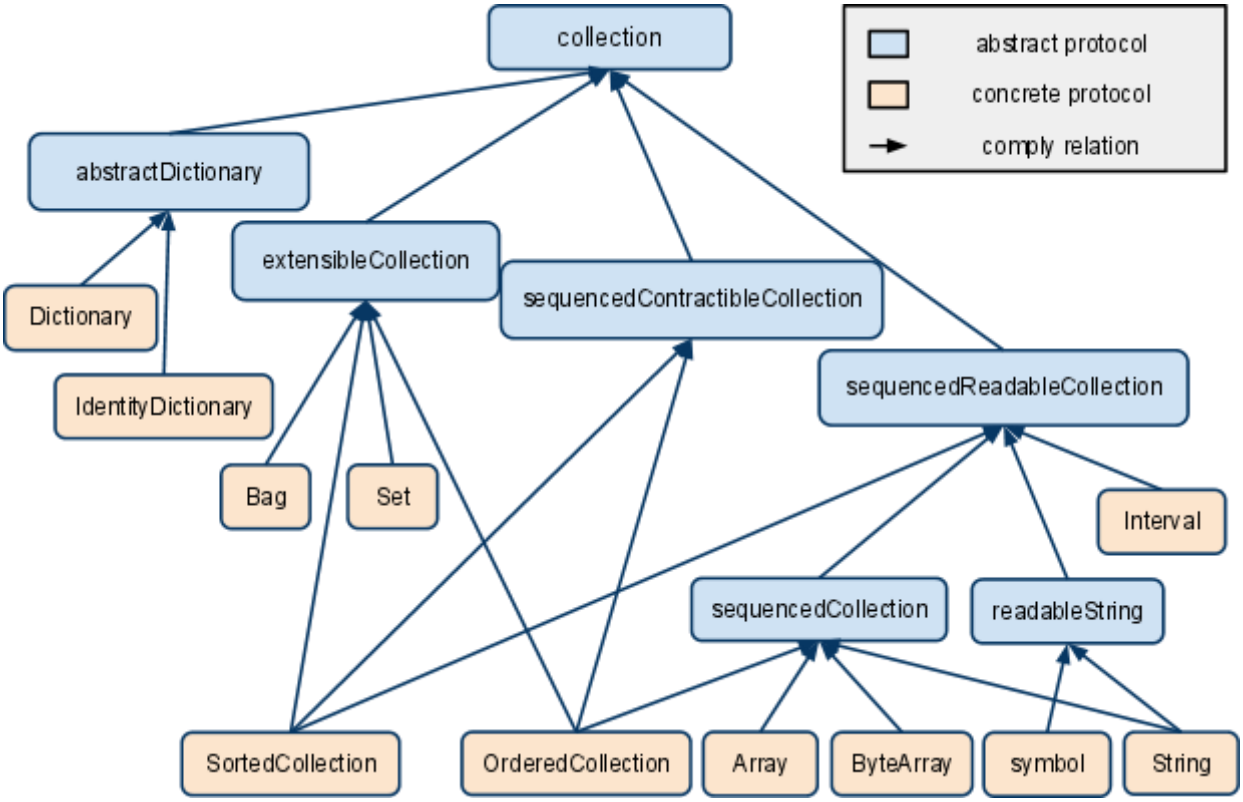

Figura 66 - Protocolo ANSI correspondiente a la jerarquía de Collection

Las flechas indican la relación de conformidad entre protocolos. Esto quiere decir que el protocolo origen de la relación cumple con el protocolo destino de la relación. Por ejemplo: el protocolo *abstractDictionary* cumple con el protocolo *collection*.
De igual manera se define un protocolo para la creación de instancias de la jerarquía de Collection. La Figura 67 muestra las relaciones entre estos protocolos.

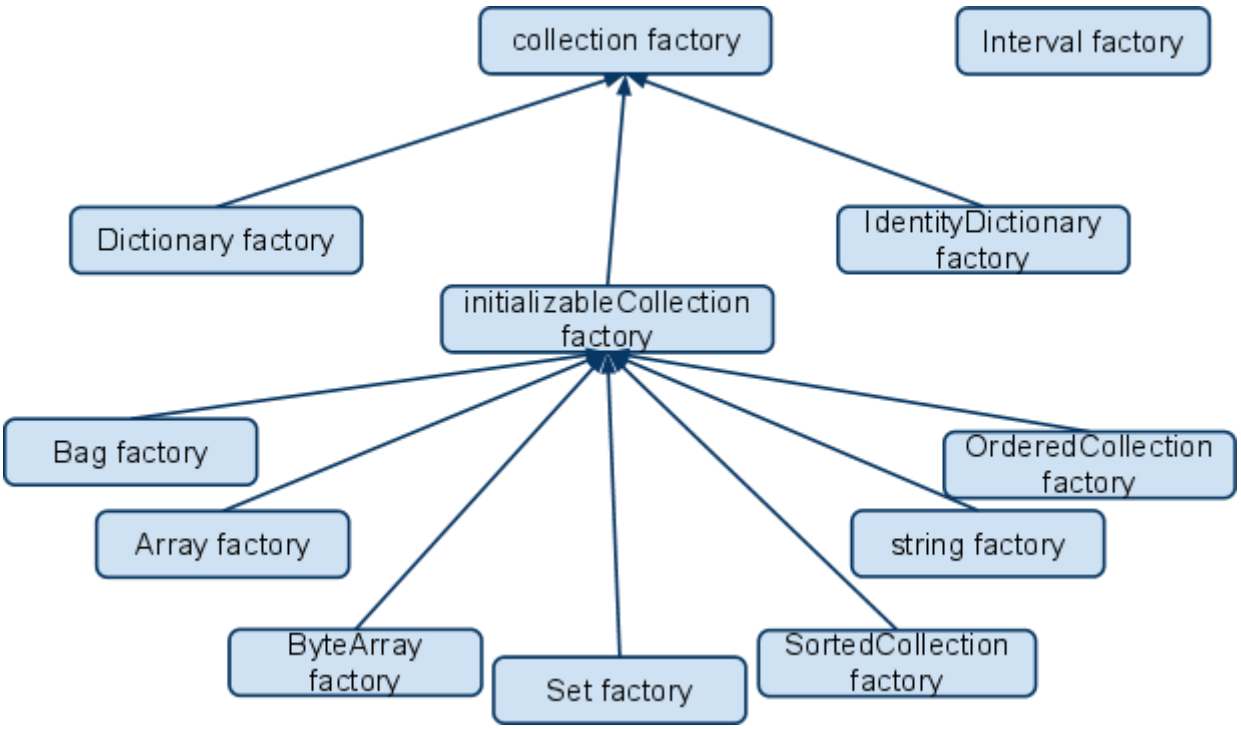

Figura 67 - Protocolo ANSI correspondiente a la creación de instancias de la jerarquía Collection

B.3 Mensajes del protocolo Collection del ANSI no implementados en Pharo

Los mensajes que a continuación se enumeran, no serán implementados en el nuevo modelo de Collection, y por lo tanto, tampoco se incluirán en los tests del protocolo de Collection del ANSI.
Se decidió no implementar estos mensajes ya que los mismos no eran relevantes para el objetivo de este trabajo.

Mensajes de Collection
No son implementados los siguientes mensajes:

- `#rehash`

Mensajes de SequenceableReadableCollection
No son implementados los siguientes mensajes:

- `#copyReplacing:withObject:`
- `#copyReplaceFrom:to:withObject:`
- `#from:to:keysAndValuesDo:`

Mensajes de Magnitude
No son implementados:

- `#between:and:`
- `#max:`
- `#min:`

Mensajes de SequencedCollection
No son implementados:

- `#replaceFrom:to:withObject:`

Mensajes de SequencedContractibleCollection
No son implementados:

- `#removeAtIndex:`
  Sin embargo, es implementado el mensaje `#removeAt:` que tiene el mismo comportamiento. Por lo tanto se utilizará `#removeAt:` en reemplazo de `#removeAtIndex:`

Mensajes de OrderedCollection
No son implementados:

- `#addAll:after:`
- `#addAll:afterIndex:`
- `#addAll:before:`
- `#addAll:beforeIndex:`

Anexo C

Jerarquía de Collections de Pharo

La Figura 68 muestra en un diagrama de clases la jerarquía de Collections de Pharo. En el mismo se incluyen únicamente las clases correspondientes al protocolo ANSI de Collections.

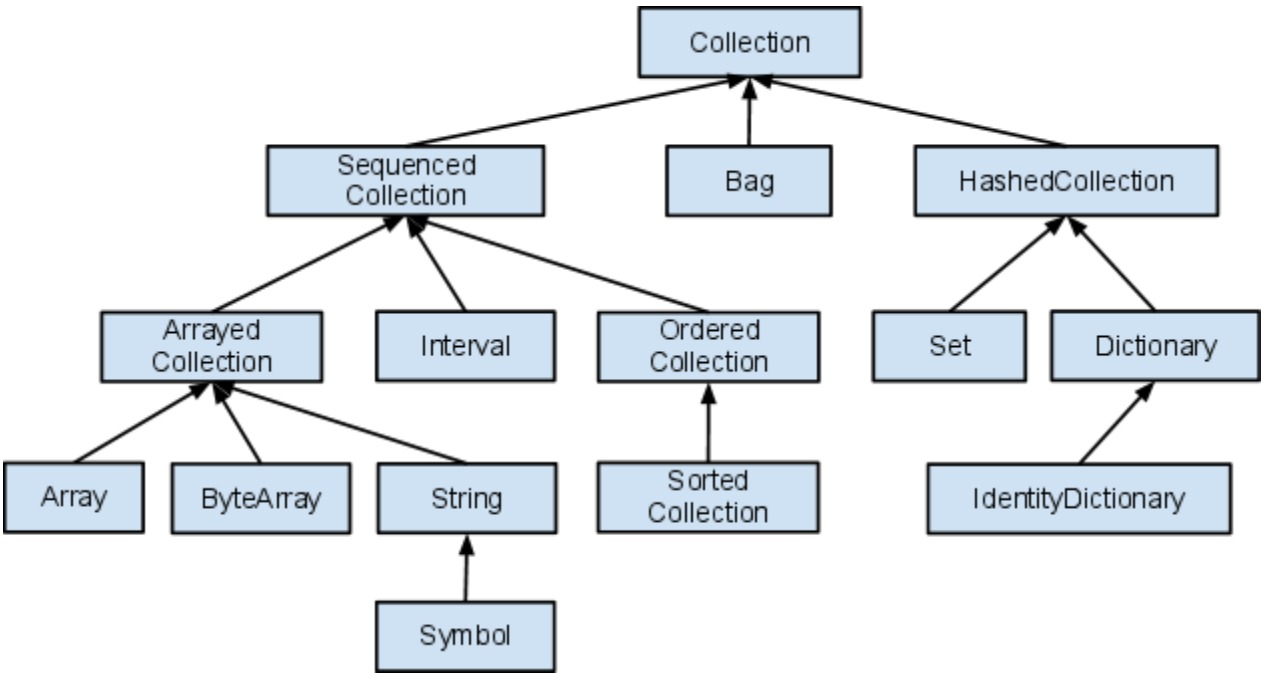

Figura 68 - Jerarquía de Collection de Pharo